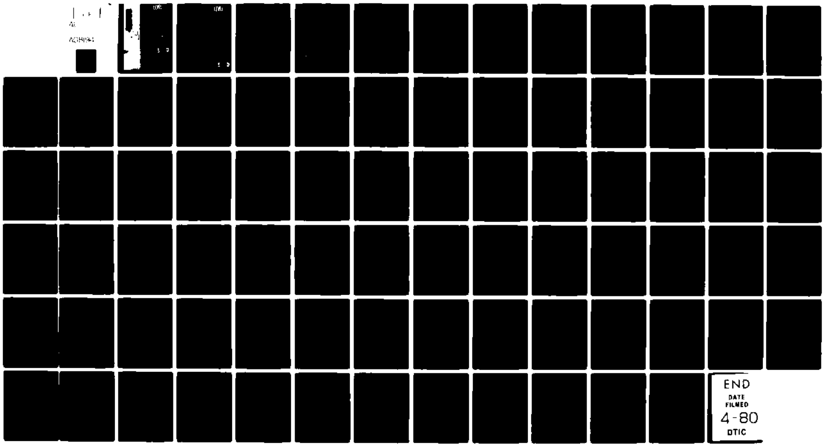
AD-A081 941 TEXAS UNIV AT AUSTIN CENTER FOR CYBERNETIC STUDIES F/G 12/2
COMPREHENSIVE COMPUTER EVALUATION AND ENHANCEMENT OF MAXIMUM FL--ETC(U)
OCT 79 F GLOVER, D KLINGMAN, J MOTE N00014-78-C-0222
UNCLASSIFIED CCS-RR-356 NL



END
DATE
FILMED
4-80
DTIC

ADA081941

(12) LEVEL II

B.S.

# CENTER FOR CYBERNETIC STUDIES

The University of Texas
Austin, Texas 78712

80 3 17 029

12 LEVEL II

9 RESEARCH REPORT 356

6

# COMPREHENSIVE COMPUTER EVALUATION AND ENHANCEMENT OF MAXIMUM FLOW ALGORITHMS.

by

10 Fred Glover*
Darwin Klingman**
John Mote***
David Whitman****

14 CCS-RR-356

15 N00014-78-C-0222, DOT-OS-70074

11 OCT [redacted] 79

12 79

* Professor of Management Science, University of Colorado, Boulder, CO 80309.

** Professor of Operations Research and Computer Sciences, The University of Texas at Austin, BEB 608, Austin, TX 78712.

*** Systems Analyst, Analysis, Research, and Computation, Inc., P.O. Box 4067, Austin, TX 78765.

**** Systems Analyst, Analysis, Research, and Computation, Inc., P.O. Box 4067, Austin, TX 78765.

This research was partially funded by the U.S. Department of Transportation, Contract No. DOT-OS-70074 and by the Office of Naval Research, Contract No. N00014-78-C-0222. Reproduction in whole or in part is permitted for any purpose of the U.S. Government.

CENTER FOR CYBERNETIC STUDIES

A. Charnes, Director
Business-Economics Building 203E
The University of Texas
Austin, TX 78712

512-471-1821

406197

## ABSTRACT

For a number of years the maximum flow network problem has attracted the attention of prominent researchers in network optimization. Since the ground-breaking work of Ford and Fulkerson, a variety of algorithms featuring good "worst-case" bounds have been proposed for this problem. Surprisingly though, there has been almost no empirical evaluations of these algorithms.

The primary purpose of this study was to refine and streamline all major classes of maximum flow algorithms using the recent developments in network labeling and data organization techniques. To safeguard against being swayed too heavily by preliminary analyses (and past experience in other network settings), IT HAS ~~we~~ implemented more than one type of data structure and associated processing techniques for most of the algorithms. Additionally, ~~we tested~~ the resulting codes on four distinct problem topologies. HAS BEEN TESTED.

During the course of our investigation we examined the two most widely heralded general classes of algorithms for maximum flow network problems--the label tree and referent algorithms. Over 50 codes were developed and at least partially tested for these methods. In the process we also developed and tested a new member of the referent class of algorithms, called the implicit referent method, which proved far more effective than all others.

In addition, we investigated a third type of approach which constitutes a special purpose variant of the primal simplex method. Previously, researchers have neglected primal methods in favor of more classical labeling types of algorithms, first because the classical methods were obvious and "natural", and second because simple choice rules yield good worst-case bounds. Over twenty implementations of our proposed variant of the primal method were tested utilizing alternative starts, pivot choice rules, and update techniques. Our computational results show that the new primal simplex variant is fastest on two of the four problem topologies and is second to the sub-referent method on the other two problem topologies. In addition, the primal simplex variant requires approximately one-third the computer memory required by other algorithms and lends itself more readily to an efficient in-core out-of-core implementation.

## 1.0 INTRODUCTION

For a number of years the maximum flow network problem has attracted the attention of prominent researchers in network optimization. Since the ground-breaking work of Ford and Fulkerson [9, 15, 16, 17, 18], a variety of algorithms [1, 5, 6, 11, 12, 13, 14, 24, 25, 27, 28, 30, 31, 34] featuring good "worst-case" bounds have been proposed for this problem. Surprisingly though, there have been almost no empirical evaluations of these algorithms.

Cheung [6] recently conducted the first significant computational investigation of maximum flow methods, testing several of the major approaches. Although an important step in the right direction, Cheung's implementations employ methodology and data structures originating at least a dozen years ago [5].

In the past decade, however, advances in network implementation technology have been dramatic. Sophisticated labeling techniques and more effective data structures have (a) decreased total solution time and/or (b) reduced computer memory requirements [2, 4, 10, 19, 20, 21, 22, 23, 32, 33]. As a result, widely held beliefs about which algorithms are best for particular problem classes have been steadily challenged and in some cases completely overturned [10, 19, 21, 26, 32, 33]. This study likewise discloses several misconceptions about maximum flow algorithms whose challenge was overdue.

One of our primary purposes, therefore, has been to design and test maximum flow implementations that make the most effective use of the recent developments in network labeling and data organization techniques. To safeguard against being swayed too heavily by preliminary analyses (and past experience in other network settings), we implemented more than one type of data structure and associated processing techniques for most of the algorithms tested. Additionally, we tested the resulting codes on four distinct problem topologies.

During the course of our investigation we examined the two most widely heralded general classes of algorithms for maximum flow network problems--the label tree and referent algorithms. Over 50 codes were developed and at least partially tested for these methods. In the process we developed and tested a new member of the referent class of algorithms, called the sub-referent method, which proved far more effective than all others.

In addition, we investigated a third type of approach which constitutes a special-purpose variant of the primal simplex method. Previously, researchers have neglected primal methods in favor of more classical labeling types of algorithms, first because the classical methods were obvious and "natural", and second because simple choice rules yield good worst-case bounds for these methods. Recently, Cunningham [7, 8] has partly removed the theoretical bias against primal simplex maximum flow methods by deriving a computational bound for one of its variants (different from the one we propose in this study). Although this theoretical bound is not nearly as good as those for label tree and referent algorithms, practical experience in the network area over the past decade [4, 20, 21, 33] argues strongly for testing a derivative of the primal simplex method which has proved highly robust and effective. We tested over twenty implementations of our proposed variant of the primal method, utilizing alternative starts, pivot choice rules, and update techniques.

Two of the approximately 70 algorithmic implementations of label tree, referent and primal methods stand out far above all the rest. One of these is an implementation of the new sub-referent algorithm, which is dramatically faster than all contenders on most of the problems tested. The other method that stands above the rest of the field is an implementation of the new primal simplex variant, which is fastest on two of the problem topologies and is second in efficiency to the sub-referent method on the other two problem topologies. In addition, the primal simplex variant requires approximately one-third the computer memory required by other algorithms, and lends itself more readily to an efficient in-core out-of-core implementation.

## 2.0 PROBLEM DEFINITION

Let $G(N,A)$ be a directed network consisting of a finite set $N$ of nodes and a finite set $A$ of arcs, where each arc $k \in A$ may be denoted as an ordered pair $(u,v)$ (referring to the fact that the arc is directed from node $u \in N$ to node $v \in N$). Associated with each arc $k = (u,v)$ is a flow variable $x_k$ and an upper bound or capacity coefficient $u_k$. Additionally, two specific nodes $s \in N$ and $t \in N$ are called the source and terminal, respectively.

The maximum flow network problem may be stated as:

$$\text{Maximize} \quad x_0 \tag{1}$$

subject to:

$$-\sum_{k \in FS(s)} x_k + \sum_{k \in RS(s)} x_k = -x_0 \tag{2}$$

$$-\sum_{k \in FS(t)} x_k + \sum_{k \in RS(t)} x_k = x_0 \tag{3}$$

$$-\sum_{k \in FS(i)} x_k + \sum_{k \in RS(i)} x_k = 0, \; i \in N - \{s,t\} \tag{4}$$

$$0 \leqq x_k \leqq u_k, \text{ for all } k \in A \tag{5}$$

$$x_0 \geqq 0 \tag{6}$$

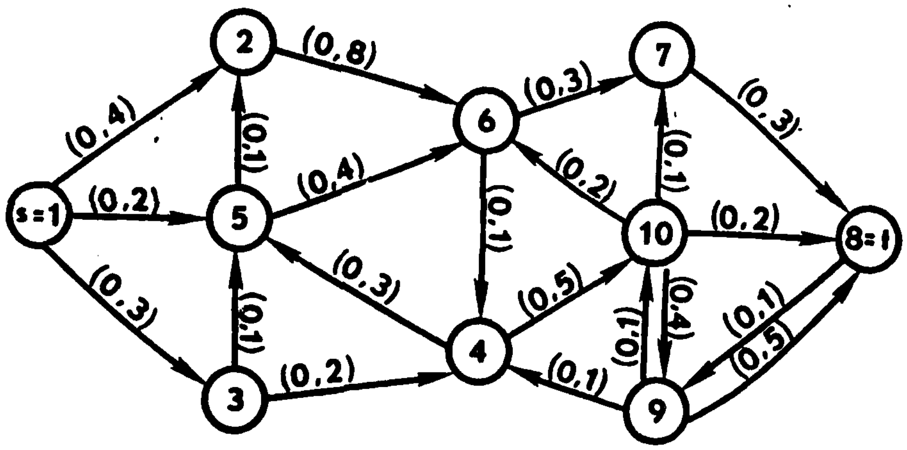
where $FS(i) = \{k:k = (i,j) \in A\}$ and $RS(i) = \{k:k = (j,i) \in A\}$. $FS(i)$ is called the *forward star* of node $i$ and is the subset of arcs that originate at node $i$. Correspondingly, $RS(i)$ is called the *reverse star* of node $i$ and is the subset of arcs that terminate at node $i$. The *complete star* of a node is defined as the union of its forward and reverse stars. Where there is no danger of confusion, we will also sometimes refer to the nodes that are endpoints of a (forward, reverse, or complete) star as elements of the star itself. It is standardly assumed that $RS(s)$ and $FS(t)$ are empty, and hence sums of flows over arcs in these sets are often not included in (2) and (3).

In the following sections, depending upon the algorithm and the data structures employed, it si sometimes preferable to rewrite problem (1)-(6) in alternative equivalent forms. In each case, the logic behind the restructuring of the problem is given.

To unify our discussions, the example network shown in Figure 1 is used throughout to illustrate various aspects of alternative implementations. In this example, the source is node 1 and the terminal is node 8. Numbers enclosed in parentheses on the arcs specify the lower and upper bounds $(0,u_k)$.

FIGURE 1

EXAMPLE NETWORK



## 3.0 EXPERIMENTAL DESIGN

### 3.1 Overview

Alternative implementation methods are evaluated in this study by solving a diverse set of randomly generated maximum network flow problems using the same computer (a CDC 6600) and the same FORTRAN compiler (MNF). All codes were executed during periods of comparable demand for computer use and were implemented by the same systems analysts with no attempt to exploit machine hardware characteristics.

Even with these safeguards, minor differences between the solution times of any two codes, for a single test run of each, are of questionable significance. For this reason, most of the problem networks were solved five times (i.e., for five diff ·ent source-terminal pairs) and the median solution time reported. Each code makes use of a real-time clock routine supplied by the computer vendor. Such routines can be accessed by a FORTRAN subroutine call and are generally accurate to two decimal places.

Problem data are input ("read in") to each code in exactly the same fashion (arc by arc, in the same "random" order). However, the codes use different data structures to store the networks. The *total solution time* records the elapsed time after input of the network and prior to the output of its solution. This includes the time required to initialize the function arrays. However, codes that are able to store the networks as originally input have an obvious advantage under this timing procedure. Consequently, a second solution statistic for each problem measures only the problem *optimization time* and disregards the time required to arrange the problem data in the function arrays and to retrieve the solution in a suitable output form.

## 3.2 Test Problems

The primary objective of this research effort was to design a solution algorithm that can be used to solve the large-scale maximum flow problems that arise in the design and analysis of real-world transportation systems. In order to evaluate the solution capabilities of the numerous algorithmic variants and refinements that were studied, a data base of test problems was required. The design of this data base was the focus of a considerable amount of effort. A number of researchers in the network area were contacted regarding their opinions on problem structures most appropriate for investigation.

As a result of this analysis, four different classes of test problems were selected for the data base. Many members of each class of problems, with varying numbers of nodes and arcs, and varying arc capacities, were generated in order to analyze the effects of the problem dimensions on algorithmic solution capabilities. Altogether, over one hundred and fifty test problems were included in the data base.

Each of the four major classes of test problems will be described briefly. A uniform probability distribution was used in all instances to randomly select items such as nodes and upper bounds.

The first and simplest class of test problems consists of unstructured or "random" networks. Such a network is constructed by initially identifying the node set N (whose elements may be assumed to be numbered from one to $|N|$). The set of arcs, A, is generated by successively selecting ordered pairs of nodes, $u \in N$ and $v \in N - \{u\}$ thus creating an arc (u,v) for each pair selected. Multiple arcs directed from node u to node v are not allowed in this, or the other three, classes of test problems. The integer upper bounds, or arc capacities, are selected within a pre-defined interval. The source node, s, and the terminal node, t, are randomly chosen from the elements of N.

Due to the simplicity of the arc generation process, this class of problems possesses no specific underlying structure, and hence is referred to as the *random* class. Twelve different sets of problem dimensions, R1, R2,...,R12, were selected for this class, each containing five different problems. The specific problem dimensions are provided in Table I. Random problems were included in the study because they represent the closest analogy to test problems used by Cheung [6] and because most maximum flow literature makes no reference to any particular problem structure.

The second class of problems is called the *multi-terminal random* class. Unlike a random network, a multi-terminal random network possesses a small degree of underlying structure. The source and terminal nodes for these problems actually play the role of master source and master terminal nodes. This results by assigning infinite capacities to all arcs incident upon these two nodes, so that all nodes in the (forward) star of the source node serve as "effective sources," and all nodes in the (reverse) star of the terminal node serve as "effective terminals."

The overall impact of this slight generalization of the random network structure is to create a problem that simulates a true multiple source and

TABLE I

RANDOM PROBLEM
SPECIFICATIONS

| PROBLEM | \|N\| | \|A\| | ARC CAPACITY RANGE |
|---|---|---|---|
| R1 | 250 | 1250 | 1-100 |
| R2 | 250 | 1875 | 1-100 |
| R3 | 250 | 2500 | 1-100 |
| R4 | 500 | 2500 | 1-100 |
| R5 | 500 | 3750 | 1-100 |
| R6 | 500 | 5000 | 1-100 |
| R7 | 750 | 3750 | 1-100 |
| R8 | 750 | 5825 | 1-100 |
| R9 | 750 | 7500 | 1-100 |
| R10 | 1000 | 5000 | 1-100 |
| R11 | 1000 | 7500 | 1-100 |
| R12 | 1000 | 10000 | 1-100 |

multiple terminal network. While the objective of a random network problem is to determine the maximum flow from the source node s to the terminal node t, the objective of a multi-terminal random network problem is to determine the maximum flow from the set of effective source nodes $\{v|(s,v) \varepsilon A\}$ to the set of effective terminal nodes $\{u|(u,t) \varepsilon A\}$. It is important to distinguish between "true" and "simulated" problems because prior knowledge that a problem contains multiple sources and terminals can be used to re-design an algorithm to make it more effective for this situation. One goal of this study was to pose maximum flow problems of alternative structures without "informing" the algorithm in advance what those structures were.

Twelve different sets of problems, MR1, MR2,...,MR12, were selected for the multi-terminal random class, and five problems were generated for each set. All problems from a given set share the same problem dimensions.

Specific problem dimensions are indicated in Table II.

Multi-terminal random problems were included in the study because some researchers have conjectured that referent algorithms are naturally designed to work well on them. (That is, referent methods are innately less blind to this type of model structure.) Our computational results support this point of view.

TABLE II

MULTI-TERMINAL RANDOM PROBLEM SPECIFICATIONS

| PROBLEM | \|N\|* | \|A\| | AVERAGE NO. OF ARCS INCIDENT ON EACH MASTER SOURCE (TERMINAL) | ARC CAPACITY RANGE** |
|---|---|---|---|---|
| MR1 | 250 | 1250 | 5.0 | 1-100 |
| MR2 | 250 | 1875 | 7.5 | 1-100 |
| MR3 | 250 | 2500 | 10.0 | 1-100 |
| MR4 | 500 | 2500 | 5.0 | 1-100 |
| MR5 | 500 | 3750 | 7.5 | 1-100 |
| MR6 | 500 | 5000 | 10.0 | 1-100 |
| MR7 | 750 | 3750 | 5.0 | 1-100 |
| MR8 | 750 | 5825 | 7.5 | 1-100 |
| MR9 | 750 | 7500 | 10.0 | 1-100 |
| MR10 | 1000 | 5000 | 5.0 | 1-100 |
| MR11 | 1000 | 7500 | 7.5 | 1-100 |
| MR12 | 1000 | 10000 | 10.0 | 1-100 |

*There were five master source nodes and five master terminal nodes.

**Excluding arcs entering or leaving source and terminal nodes.

The third class of problems introduces additional structure into the network. This problem class is called the *transit grid* class. The source and terminal nodes again serve as a master source and a master terminal, as in the multi-terminal random problems, implicitly creating a set of effective sources and effective terminals. All nodes other than s and t are referred to as grid nodes, which can be viewed as arranged in a rectangular grid of r rows and c columns. Every adjacent pair of grid nodes is connected by two oppositely directed arcs whose capacities are selected from a pre-defined interval.

Like the multi-terminal random class, this class of problems simulates multiple source and multiple terminal networks (and the algorithms are not amended to capitalize on this fact). Unlike the random and the multi-terminal random networks, the additional structure of the transit grid networks closely resembles that arising in urban transit planning networks. In this setting, the grid structure captures the form of transportation routes in the greater suburban area. Source nodes represent major transit exchanges or vehicle storage facilities and terminal nodes correspond to collection nodes which are connected to key demand points within the city.

Eight different sets of problem dimensions were chosen for the transit grid class. Again, five different problems were generated for each set of dimensions. These sets of transit grid problems are referred to as TG1, TG2,...,TG8. Table III contains the specific problem dimensions for these sets.

The final class of test problems possesses the most elaborate structure. This class consists of totally dense, acyclic networks involving an even number of nodes. That is, every pair of nodes is connected by an arc directed from the node with the smaller node number to the node with the larger node number. The capacity of the arc (u,v) is 1 if $v > u + 1$ and is $1 + (u - \frac{|N|}{2})^2$ if $v = u + 1$. Node 1 is the source node and node $|N|$ is the terminal node.

Although somewhat artificial, this class of problems was included because it was expected to require a large number of iterations (starting from a zero flow initial state) since the optimal solution is obtained when the flow on every arc in the network is at its upper bound. For obvious reasons,

TABLE III

TRANSIT GRID PROBLEM SPECIFICATIONS

| PROBLEM | \|N\|* | \|A\| | AVERAGE NO. OF ARCS INCIDENT TO EACH MASTER SOURCE (TERMINAL) | ARC CAPACITY RANGE** |
|---|---|---|---|---|
| TG1 | 235 | 1240 | 40 | 1-100 |
| TG2 | 235 | 1640 | 80 | 1-100 |
| TG3 | 410 | 2120 | 60 | 1-100 |
| TG4 | 410 | 2720 | 120 | 1-100 |
| TG5 | 635 | 3200 | 80 | 1-100 |
| TG6 | 635 | 4000 | 160 | 1-100 |
| TG7 | 910 | 4480 | 100 | 1-100 |
| TG8 | 910 | 5480 | 200 | 1-100 |

*Including five master source nodes and five master terminal nodes.

**Excluding arcs entering or leaving master source and master terminal nodes.

this class is referred to as the *hard* class. Five problems, H1, H2,..., H5, were considered differentiated by their dimensions. Table IV presents the relevant parameters.

TABLE IV

HARD PROBLEM SPECIFICATIONS

| PROBLEM | $\|N\|$ | $\|A\|$ | ARC CAPACITY RANGE |
|---|---|---|---|
| H1 | 20 | 190 | 1-82 |
| H2 | 40 | 780 | 1-362 |
| H3 | 60 | 1770 | 1-782 |
| H4 | 80 | 3160 | 1-1522 |
| H5 | 100 | 4950 | 1-2402 |

## 4.0 PRIMAL SIMPLEX VARIANT

### 4.1 Algorithm and Implementation Overview

To describe our specialization of the primal simplex method for the maximum flow network problem, it is useful to rewrite problem (1)-(6) in an equivalent form that makes it possible to isolate a particular basis structure. The equivalent formulation is:

Maximize $x_o$ (7)

subject to:

$$-\sum_{k\epsilon FS(s)} x_k + \sum_{k\epsilon RS(s)} x_k + x_o = 0 \quad (8)$$

$$-\sum_{k\epsilon FS(t)} x_k + \sum_{k\epsilon RS(t)} x_k - y = 0 \quad (9)$$

$$-\sum_{k\epsilon FS(i)} x_u + \sum_{k\epsilon RS(i)} x_u = 0 \quad i\epsilon N-\{s,t\} \quad (10)$$

$$\vdots$$

$$-x_o + y = 0 \qquad (11)$$

$$0 \leq x_k \leq u_k \qquad k \in A \qquad (12)$$

$$0 \leq x_o,\ y \qquad (13)$$

The preceding formulation arises by augmenting the original network G(N,A) by an additional node associated with equation (11) and two additional arcs associated with the variables $x_o$ and y. Letting d denote the additional node and $G(\overline{N},\overline{A})$ denote the full associated network, then $\overline{N} = N \cup \{d\}$ and $\overline{A} = A \cup \{(t,d),\ (d,s)\}$.

Problem (7)-(13) constitutes a special circulation format for problem (1)-(6). Obviously, problem (1)-(6) could have been circularized more compactly by simply moving the right hand side of (2) and (3) to the left hand side, thereby implicitly adding an arc from the terminal to the source. The reason for using instead the format of (7)-(13) will soon become apparent.

In our variant of the primal simplex method, the basis tree $T(\overline{N},\overline{A}_T)$ is distinguished by the choice of node d as the root. Furthermore, without loss of generality, we may assume that arcs (d,s) and (t,d) are basic and are thus in $\overline{A}_T$. Consequently, nodes s and t always hang from the root d. This organization enables the remaining nodes $\overline{N}$ - {d,s,t} to be partitioned into two subsets: those hanging below node s and those hanging below node t. For ease of discussion we refer to nodes on the *s-side* and *t-side* of the tree. (This node partitioning corresponds to a *cut* in graph terminology.)

Two ways of storing the original problem data were tested for our variant of the primal simplex maximum flow algorithm. The first, called the *random* form, uses three |A| length lists, FROM, TO, and CAP, to sotre the original problem data without arranging them in any particular order. This effectively restricts the algorithm to simple sequential processing of the arcs, since the data structure does not allow individual forward or reverse stars to be accessed efficiently. The random form data structure is illustrated in Figure 2 for the problem given in Figure 1. The list labeled ARC in Figure 2 is provided simply to facilitate interpretation, and is not used in computer implementations of the random form data structure.

FIGURE 2

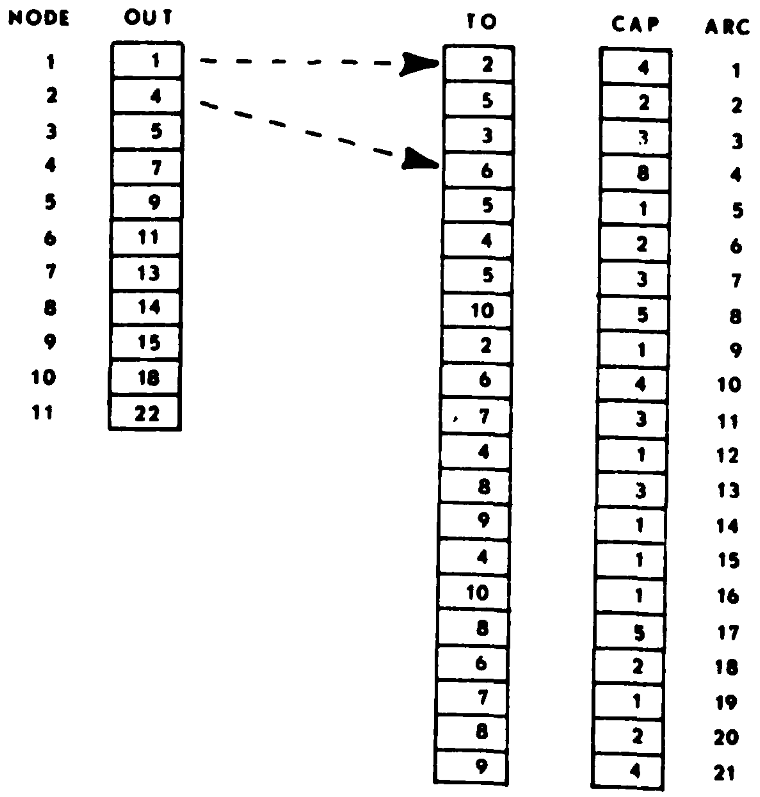
RANDOM FORM

| ARC | FROM | TO | CAP |
|---|---|---|---|
| 1 | 1 | 2 | 4 |
| 2 | 2 | 6 | 8 |
| 3 | 5 | 2 | 1 |
| 4 | 5 | 6 | 4 |
| 5 | 6 | 7 | 3 |
| 6 | 6 | 4 | 1 |
| 7 | 10 | 6 | 2 |
| 8 | 7 | 8 | 3 |
| 9 | 10 | 7 | 1 |
| 10 | 10 | 8 | 2 |
| 11 | 9 | 8 | 5 |
| 12 | 8 | 9 | 1 |
| 13 | 9 | 10 | 1 |
| 14 | 10 | 9 | 4 |
| 15 | 9 | 4 | 1 |
| 16 | 4 | 10 | 5 |
| 17 | 3 | 4 | 2 |
| 18 | 4 | 5 | 3 |
| 19 | 1 | 5 | 2 |
| 20 | 1 | 3 | 3 |
| 21 | 3 | 5 | 1 |

The second data structure used in the testing of the primal maximum flow algorithm is called the *forward star* form. With this data structure the arcs must be sorted according to common origin nodes, storing arcs of a given forward star in contiguous locations in the arc data lists. Since each arc in a forward star has the same from-node, and since the arcs are groups by forward stars, it is unnecessary to store the from-node for every node in the network (which would require an |A| length FROM array, as in the random form). Rather, it is possible to use an |N| length list, OUT, which points to the location in the arc data of the first arc in each forward star. In addition to this |N| length list, the forward star form uses two |A| length lists, TO and CAP. This data structure is illustrated in Figure 3. The lists labeled NODE and ARC in Figure 3 serve simply as guides to aid understanding.

FIGURE 3

FORWARD STAR FORM



| NODE | OUT |
|---|---|
| 1 | 1 |
| 2 | 4 |
| 3 | 5 |
| 4 | 7 |
| 5 | 9 |
| 6 | 11 |
| 7 | 13 |
| 8 | 14 |
| 9 | 15 |
| 10 | 18 |
| 11 | 22 |

| TO | CAP | ARC |
|---|---|---|
| 2 | 4 | 1 |
| 5 | 2 | 2 |
| 3 | 3 | 3 |
| 6 | 8 | 4 |
| 5 | 1 | 5 |
| 4 | 2 | 6 |
| 5 | 3 | 7 |
| 10 | 5 | 8 |
| 2 | 1 | 9 |
| 6 | 4 | 10 |
| 7 | 3 | 11 |
| 4 | 1 | 12 |
| 8 | 3 | 13 |
| 9 | 1 | 14 |
| 4 | 1 | 15 |
| 10 | 1 | 16 |
| 8 | 5 | 17 |
| 6 | 2 | 18 |
| 7 | 1 | 19 |
| 8 | 2 | 20 |
| 9 | 4 | 21 |

In addition to the lists needed to store the original problem data, these implementations of the primal simplex maximum flow algorithm require six $|N|$ length lists to efficiently store and update the basis tree and the corresponding primal and dual solutions. These lists are: the predecessor node (PN) [4], predecessor arc (PA) [2], thread (THREAD) [4], depth (DEPTH) [3], node potential (POT) [3], and net capacity (NETCAP) [3]. The predecessor node is an "upward" pointer in the basis tree. It identifies the unique node directly above each node in the basis tree (except the root). The predecessor arc is the arc number of the basic arc connecting a node to its predecessor node and provides access to the original problem data. The thread, by contrast, is a circular list that links the nodes of the basis tree in a top to bottom, left to right fashion, enabling all nodes in any subtree to be traced in unbroken succession. The depth of the node is simply the number of arcs on the unique path in the basis tree from the node to the root, while the node potentials are the dual variables associated with the current basis tree. Finally, the net capacity is the amount of allowable flow change on the predecessor arc in the direction from a node to its predecessor node. That is, if the predecessor arc is pointed down (up) in the basis tree, then the net capacity of the arc is simply the current flow on the arc (upper bound minus current flow). This unorthodox way of storing relevant solution data for the problem actually simplifies the solution procedure. An example of a feasible basis tree is given in Figure 4. The associated $|N|$ length lists are given in Figure 5. The list labeled NODE in Figure 5 is provided simply to facilitate interpretation and is not used in computer implementation.

By associating a dual variable $\pi_i$ with each node $i \in \bar{N}$, the complementary slackness property of linear programming implies the following:

$$-\pi_d + \pi_s = 1 \tag{14}$$

$$\pi_d - \pi_t = 0 \tag{15}$$

$$-\pi_i + \pi_j = 0 \quad \text{for all } (i,j) \in \bar{A}_T - \{(t,d), (d,s)\} \tag{16}$$

FIGURE 4
FEASIBLE BASIS TREE

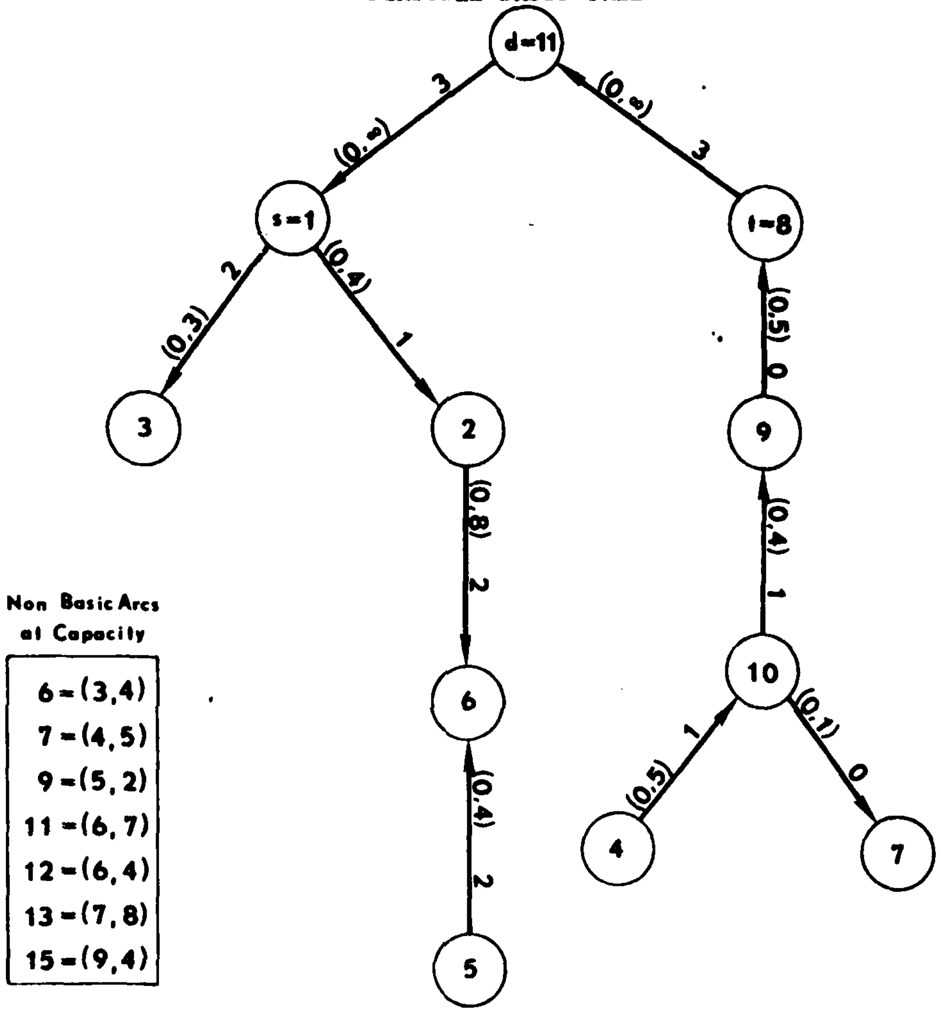


FIGURE 5
NODE LENGTH LISTS ASSOCIATED WITH FIGURE 4

| NODE | PN | PA | THREAD | DEPTH | POT | NETCAP |
|---|---|---|---|---|---|---|
| 1 | 11 | 23 | 3 | 1 | 1 | 3 |
| 2 | 1 | 1 | 6 | 2 | 1 | 1 |
| 3 | 1 | 3 | 2 | 2 | 1 | 2 |
| 4 | 10 | 8 | 7 | 4 | 0 | 4 |
| 5 | 6 | 10 | 8 | 4 | 1 | 2 |
| 6 | 2 | 4 | 5 | 3 | 1 | 2 |
| 7 | 10 | 19 | 11 | 4 | 0 | 0 |
| 8 | 11 | 22 | 9 | 1 | 0 | ∞-3 |
| 9 | 8 | 17 | 10 | 2 | 0 | 5 |
| 10 | 9 | 21 | 4 | 3 | 0 | 3 |
| 11 |  |  | 1 | 0 | 0 |  |

Since redundancy allows the value of any one of the $\pi_i$ to be set arbitrarily, we elect to set $\pi_d = 0$. This choice, made natural by the choice of d as root, yields a solution to (14)-(16) such that $\pi_i = 1$ if node i is on the s-side of the basic tree and $\pi_i = 0$ if i is on the t-side of the basis tree. This property is illustrated in Figure 5.

It may be remarked that the node potentials associated with an optimal basis tree also provide information regarding the minimum cut problem. Specifically, the potentials indicate which side of the cut the node lies on; i.e., node i is on the s-side of the cut if and only if $\pi_i = 1$. Arcs whose capacities define the capacity of the cut are also identified by the node potentials: the capacity of arc (i,j) is included in the capacity of the cut if $\pi_i \neq \pi_j$. Additionally, the arc is directed from s-side (t-side) to t-side (s-side) if $\pi_i = 1$ (0) and $\pi_j = 0$ (1).

In view of the foregoing observations, the general form of the steps for our variant of the primal simplex algorithm for the maximum flow problem may be summarized as follows:

STEP 1: [INITIALIZATION] Select an initial feasible basis tree rooted at node d.

STEP 2: [DUAL SOLUTION] Determine the potential $\pi_i$ of each node, according to the preceding observations, that results by setting $\pi_d = 0$.

STEP 3: [ENTERING ARC] Select a non-basic arc e from node i to node j such that (a) $\pi_i = 1$, $\pi_j = 0$, and $x_e = 0$, or
(b) $\pi_i = 0$, $\pi_j = 1$, and $x_e = u_e$.
If no such arc exists, stop. The optimal solution has been found.

STEP 4: [LEAVING ARC] If $x_e = 0$ ($x_e = u_e$), determine the maximum amount, $\delta$, that flow can be increased (decreased) on arc (i,j) by changing the flows on the unique path from node i to node j in the current basis tree. If $\delta \geq u_e$, go to STEP 6.

STEP 5: [CHANGE OF BASIS] If $x_e = u_e$, set $\delta = -\delta$. Let arc r be one of the arcs that only allows a flow change of $|\delta|$ in STEP 4. Change the flow by $\delta$ on the unique path from node i to node j in the basis tree. Change the flow on arc e by $\delta$. Replace arc r with arc e in the set of basic arcs $A_T$ and update the basis tree labels. Go to STEP 3.

STEP 6: [NO CHANGE OF BASIS] If $x_e = u_e$, set $\delta = -u_e$. Otherwise set $\delta = u_e$. Change the flow by $\delta$ on the unique path from node i to node j in the basis tree. Change the flow on arc e by $\delta$. Go to STEP 3.

The specific implementation of each step of the preceding algorithm of course plays a major role in determining the overall efficiency of a particular computer code. During the extensive testing of this primal simplex maximum flow algorithm, over twenty alternative implementations of the basic algorithm were developed. These alternatives were used to determine the impact of the choice of the starting basis (STEP 1), selection of the entering arc (STEP 3), and the selection of the leaving arc (STEP 4).

The same techniques, whose efficiency has been well-established, were used throughout to update the list structures (STEPS 2, 5, and 6). We exploited the fact that the unique basis equivalent path (for any entering arc) contains the root node d to update the flows on the arcs in this path in a single pass. This one pass update is efficiently carried out by using the predecessor node and depth functions (in conjunction with the net capacity function).

The next subsections contain descriptions of the various start and pivot strategies examined. The final subsection contains the results of our testing.

## 4.2 Starts

Four basic implementations of STEP 1 were tested during the course of this study. The first implementation is referred to as the *LIFO label-out start* procedure. The basic steps of this procedure are:

STEP 1A: [INITIALIZATION] Initialize the predecessor node function:

$$PN(d) = 0,\ PN(s) = d,\ PN(t) = d.$$

Initialize a node label function:

LABEL(d) = -1

LABEL(s) = -1

LABEL(t) = -1

LABEL(i) = 0 for all $i \in N - \{s,t\}$

Set i = s.

STEP 1B: [NODE SCAN] For each arc (i,j) in the forward star of node i, check the label status of node j. If LABEL(j) = 0, then set PN(j) = i, LABEL(j) = LABEL(i) and LABEL(i) = j.

STEP 1C: [NEXT NODE] Set i = LABEL(i). If i > 0 go to STEP 1B. Otherwise, the construction of the initial LIFO label-out tree is complete.

This implementation uses the node label function (LABEL) as a depth first sequence list. That is, each newly labeled node is placed at the front of the sequence list. Since this approach only requires a single pass through the arc data, it uses very little c.p.u. time to execute. The creation of the necessary node function values are easily incorporated into STEP 1B so that all initialization is carried out during the construction of the initial basis tree.

The initial solution constructed by the LIFO label-out start procedure exhibits the following characteristics: all arcs, basic as well as non-basic, have zero flow; all nodes except the terminal and the root are on the s-side of the basis tree; and all basic arcs, except (t,d) are directed away from the root.

The second implementation of STEP 1 is called the *FIFO label-out start* procedure. This start is identical to the LIFO label-out start except that each newly labeled node is placed at the back, instead of the front, of the sequence (LABEL) list. Thus it has the characteristics indicated for the LIFO procedure.

The primary difference between the two methods is the shape of the basis tree. FIFO, the breadth first start, tends to construct a wide, shallow initial tree whereas LIFO, the depth first start, tends to construct a narrow, deep initial tree.

The third implementation of STEP 1 that was tested is called the *balanced tree start* procedure. Unlike the other implementations, this procedure requires multiple passes of the arc data, and therefore more c.p.u. time is required to execute the start. However, this start procedure, as its name implies, attempts to balance the number of the nodes on the s-side and t-side of the basis tree, and was motivated by the fact that the only pivot eligible arcs (STEP 3) are those whose nodes are on opposite sides of the basis tree. By balancing the number of nodes on each side, we expected that the entering arc could be selected more efficiently. The steps of the balanced tree start are outlined below.

STEP 1A: [INITIALIZATION] Initialize the predecessor node function:

$$PN(d) = 0,\ PN(s) = d,\ PN(t) = d.$$

Initiate a node label function:

$LABEL(d) = -1$

$LABEL(s) = -1$

$LABEL(t) = 1$

$LABEL(i) = 0$ for all $i \in N - \{s,t\}$

Set $i = s$.

STEP 1B: [NODE SCAN CHOICE] If LABEL(i) = -1, go to STEP 1D. If LABEL(i) = 0, go to STEP 1E.

STEP 1C: [NEXT NODE] If a complete pass of the arc data fails to relabel any nodes, go to STEP 1F. Otherwise, select a node i. Go to STEP 1B.

STEP 1D: [SCAN DOWN] Set LABEL(i) = -2. For each arc (i,j) in the forward star of node i, check the label status of node j. If LABEL(j) = 0, then set PN(j) = i and LABEL(j) = -1. Go to STEP 1C.

STEP 1E: [SCAN UP] Select an arc (i,j) in the forward star of node i such that LABEL(j) = 1. If no such arc exists, go to STEP 1C. Otherwise let PN(i) = j and LABEL(i) = 1. Go to STEP 1C.

STEP 1F: [FINAL PASS] For each arc (i,j) such that LABEL(i) = 0 and $LABEL(j) \neq 0$ let PN(i) = j and LABEL(i) = 1.

The initial multiple passes through the arc data (STEPS 1C, 1D, and 1E) attempt to select as many arcs as possible that are either directed away from the root node on the s-side or toward the root node on the t-side. The final pass through the arc data allows any unlabeled nodes to be assigned to the t-side by means of arcs directed toward the root. Like the other two start procedures, all network arcs have zero initial flow. It should be noted that it is unnecessary to repeat the initial steps until a pass of the arc data fails to add another arc to the tree (STEP 1C).

The final start procedure tested, called the *modified balanced tree start*, begins the final pass (a modified STEP 1F) as soon as an initial pass (STEPS 1B-1E) adds fewer than $n_1$ nodes to the basis tree, or as soon as a total of $n_2$ nodes have been added to the basis tree.

## 4.3 Pivot Strategies

One of the most crucial aspects of any primal simplex network algorithm is the pivot strategy that is employed to select a non-basic arc to enter the basis. This corresponds to STEP 3 of the primal simplex maximum flow network algorithm. Many different pivot strategies were developed and tested during this study. This includes simple modifications of the pivot strategies that have proven successful for more general network flow problems [20, 21, 33], as well as new strategies developed from insights into the special structure of the maximum flow network problem. The level of complexity of these pivot strategies range from the simple sequential examination of forward stars to a complex candidate list with a steepest descent evaluation criteria. A brief description of each of the fundamental pivot strategies is presented.

In order for an arc to be pivot eligible its nodes must be on opposite sides of the basis tree. In addition, its flow must be at the appropriate bound. Specifically, arc k = (i,j) is a candidate to enter the basis if and only if

(a) $\pi_i = 1$, $\pi_j = 0$, and $x_k = 0$ or
(b) $\pi_i = 0$, $\pi_j = 1$, and $x_k = u_k$.

In case (a), arc k currently has no flow and is directed away from the s-side and toward the t-side. It is advantageous to attempt to increase flow on this arc by pivoting it into the basis. In case (b), arc k currently has as much flow as it can handle and it is directed away from the t-side and toward the s-side. In this instance, it appears to be advantageous to decrease flow on the arc, thus leading to a net increase in flow from the source to the terminal. For example, in Figure 4 arc (4,5) is eligible to enter the basis from its upper bound (case (b) pivot).

In a sense, the occurrence of a case (b) pivot implies that the algorithm previously made a "mistake" by putting too much flow on arc k, since at this point it appears beneficial to decrease flow on the arc. A statistical analysis of the test problems used for this study indicates that the primal simplex maximum flow network algorithm tends to concentrate on case (a) pivots. For most of the variants of the basic algorithm, over 98% of the pivots were of the case (a) type. This observation motivated the development of some specialized pivot strategies that initially concentrate on the case (a) entering arcs. This yields a two-phase (suboptimization) solution approach. During Phase I, only the case (a) arcs are allowed to enter the basis, and during Phase II, any pivot eligible arc is allowed to enter the basis.

The first class of pivot strategies is called *sequential* because the arcs are examined sequentially. The simplest sequential pivot strategy selects the first pivot eligible arc encountered to enter the basis. A two-phase sequential pivot strategy restricts its initial pivots to the case (a) type. The extent to which this restriction is made can have an impact on the overall solution efficiency of the algorithm. At one extreme, all case (b) pivots are postponed until the very end of the solution process. That is, the algorithm suboptimizes the problem by just allowing the case (a) pivots, then optimizes the problem by allowing both case (a) and case (b) pivots. Another implementation of the two-phase sequential pivot strategy restricts pivots to the case (a) type for the first $p_1$ pivots (or the first $p_2$ passes through the arc data).

The three sequential approaches are called, respectively, (1) SEQ/NS (sequential with no suboptimization), (2) SEQ/CS (sequential with complete suboptimization), and (3) SEQ/PS (sequential with partial suboptimization). Clearly, SEQ/PS is the most general sequential pivot strategy since setting $p_1 = 0(\infty)$ yields the SEQ/NS (SEQ/CS) pivot strategy.

The second group of pivot strategies are called *candidate list* strategies because they involve the use of a list of arcs that are potential candidates to enter the basis [29]. These strategies operate by restricting the choice for the entering arc to arcs contained in the candidate list. Periodically, this list must be reloaded with a fresh set of candidate arcs. The frequency of reloading the list, as well as the length of the list, affect the solution efficiency of the algorithm. Various criteria were studied to control the "quality" of the candidate list. This quality is governed by both the choice of the arcs to place in the list and the choice of the candidate arc to pivot into the basis.

Three criteria were considered in determining the best approach for (re)loading the candidate list. The first criterion is sequential. That is, arcs are loaded into the candidate list by sequentially examining the arc data. The specific implementations of the sequential criterion are labeled SEQ/NS, SEQ/CS, or SEQ/PS, depending upon whether no, complete, or partial suboptimization is desired.

The second criterion for selecting arcs to be placed in the candidate list was motivated by the fact that the amount of difficulty involved in carrying out the list updating procedure (STEPS 2, 5, and 6) depends to a large extent on the number of arcs in the unique (basis equivalent) path between the two nodes of the entering arc, which is the path on which the flow will be changed from the source to the terminal. Since the basis equivalent path (BEP) of any pivot eligible arc contains the root, the number of arcs in this path is simply the sum of the depth function values for the two endpoints of the incoming arc.

For these reasons, the second criterion used to reload the candidate list restricts the selection of candidates to arcs whose basis equivalent path contains fewer than p arcs. The choice of the cut-off value, p, is dynamic in nature. Initially p is selected to be small, but to guarantee optimality, p is eventually increased to $|N|$. The implementations of this criterion are labeled BEP/NS, BEP/CS, and BEP/PS, depending upon the level of suboptimization desired.

Computationally, the implementations of the BEP criterion is more difficult than those of the SEQ criterion. However, the depth function enables the number of arcs on the basis equivalent path of a pivot eligible arc to be determined quite readily.

The third criterion considered for selecting arcs to be placed in the candidate list is a form of the *steepest descent* criterion. Only arcs that cause a major change in the objective function are considered as candidates. Since the objective of the maximum flow network problem is simply to maximize the flow from the source to the terminal, this criterion reduces to a *largest augmentation* criterion. Each arc placed on the candidate list must allow a flow change of at least q units, where the cut-off value q is selected dynamically. The three implementations of the largest augmentation criterion are called AUG/NS, AUG/CS, and AUG/PS. Because the determination of the allowable flow change associated with a pivot eligible arc requires the complete traversal of its basis equivalent path, implementations of the AUG criterion are computationally cumbersome.

Slight generalizations of these basic criteria were used to select an entering arc from the candidate list. Depending upon the level of suboptimization, the SEQ criterion simply selects the first pivot eligible arc encountered in the candidate list, the BEP criterion selects the arc in the candidate list with the fewest arcs in its basis equivalent path, and the AUG criterion selects the arc that allows the largest flow augmentation.

Additional considerations for candidate list strategies include rules for controlling the length of the list as well as the frequency of reloading. In general, our tested strategies employ a dynamic length candidate list in which the number of elements is a function of the degree of difficulty involved in locating pivot eligible arcs. Typically, this results in an initial candidate list of twenty to fifty arcs. As optimality is approached, and the identification of pivot eligible arcs becomes harder, the length of the list is reduced to five or fewer arcs. Some of the tested strategies also used a maximum pivot counter to restrict the number of pivots between reloadings.

## 4.4 Leaving Arc Selection

Only two criteria were considered for determining the leaving arc

(STEP 4) from the collection of those that restrict the amount of flow change $\delta$ (i.e., that yield the minimum ratio in the standard simplex test for the outgoing variable). The customary choice is simply to select the first arc that qualifies, and we used this in most of the primal simplex codes developed for this study. However, we also tested a second strategy that has been proposed as a mechanism for controlling cycling. This strategy [3, 7, 8] is described in the next section.

## 4.5 Computational Testing

For the more than twenty variants of the basic primal simplex maximum flow algorithms developed and tested during this study, the principal questions we considered were:

1) What is the best starting basis? (STEP 1)
2) What is the best entering arc selection rule? (STEP 3)
3) What is the best leaving arc selection rule? (STEP 4)
4) What is the best data structure for storing the original problem data?

To some extent, the determination of the answer to one question depends on the answers to the other three. For example, when the original problem data is stored in the "random" format, it is virtually impossible to implement any pivot strategy based on processing a node's forward star.

### Choice of Leaving Arc

We first consider an appropriate choice for the leaving arc (STEP 4). As mentioned earlier, two strategies were tested in order to answer this question. The first strategy is known as the *network augmenting path* (*NAP*) rule. This strategy requires a special type of basis structure. Particularized to the maximum flow problem, the NAP basis structure may be characterized as one in which all basic arcs on the t-side of the tree have a positive net capacity. Given an initial starting basis of this form, the NAP rule assures all subsequent bases will have this form.

Of the starting procedures developed for this study, only the LIFO and FIFO label-out starts yield an initial basis tree with the necessary NAP structure.

Two implementations of the primal simplex algorithm were developed to test alternatives for selecting the leaving arc. The first code, NAP, uses the network augmenting path rule to select the leaving arc. The second code, NONAP, uses the simple "first minimum found" rule (selecting the first arc that qualifies to leave the basis). Both codes use the same LIFO label-out start procedure, sequential entering arc selection, and forward star data structure.

Table V presents the results of applying these two codes to the test problem data base. Neither code appears to be a definite winner on the random, multi-terminal, and transit grid problems, but the NONAP code outperforms the NAP code on the hard problems. The reason is simple: the special structure of the hard problem causes virtually every arc on the basis equivalent path to be binding. The NAP code selects the outgoing arc as close as possible to the terminal node, whereas the NONAP code tends to select the outgoing arc as close as possible to the entering arc. In fact, in about 90% of the nondegenerate pivots, the NONAP code selected the outgoing arc to be the same as the entering arc. This is a very easy pivot to perform since the structure of the basis tree remains unchanged. On the other hand, the NAP code selected the outgoing arc to be the entering arc on only about 5% of the nondegenerate pivots, resulting in more "hard" pivots than the NONAP code.

The reason that the NAP and NONAP codes performed basically the same on the other problem topologies may be related to the fact that the entering arcs are not as likely to be binding. Indeed, limited sampling of the test problems indicates the entering arc is binding for only 15% to 20% of the nondegenerate pivots for these other topologies.

Since the network augmenting path rule did not improve solution speeds and is not compatible with the balanced tree start procedures, all further reported computational testing is concerned with codes that use the first minimum found rule for STEP 3.

TABLE V

SOLUTION TIMES IN SECONDS OF PRIMAL METHODS ON CDC 6600

| PROBLEM | NAP | NONAP | LIFO | MODBAL | SEQCS | RANDOM |
|---|---|---|---|---|---|---|
| R1 | .09 | .10 | .09 | .10 | .08 | .11 |
| R2 | .23 | .23 | .19 | .22 | .18 | .25 |
| R3 | .22 | .23 | .19 | .22 | .16 | .21 |
| R4 | .34 | .36 | .32 | .24 | .16 | .23 |
| R5 | .53 | .47 | .38 | .49 | .33 | .47 |
| R6 | .66 | .66 | .52 | .58 | .52 | .58 |
| R7 | .40 | .42 | .40 | .32 | .27 | .36 |
| R8 | .64 | .64 | .55 | .60 | .48 | .51 |
| R9 | 1.04 | 1.00 | 1.21 | 1.03 | .84 | .93 |
| R10 | .90 | .81 | .73 | .39 | .46 | .51 |
| R11 | 1.36 | 1.22 | .85 | .68 | .69 | .77 |
| R12 | 1.99 | 1.81 | 2.32 | 1.75 | 1.45 | 1.62 |
| | | | | | | |
| MR1 | .29 | .32 | .28 | .28 | .26 | .28 |
| MR2 | .90 | 1.01 | .77 | .59 | .58 | .77 |
| MR3 | .68 | .87 | .68 | .61 | .56 | .58 |
| MR4 | .52 | .60 | .54 | .34 | .28 | .42 |
| MR5 | 1.25 | 1.18 | 1.25 | .88 | .83 | 1.04 |
| MR6 | 2.22 | 2.02 | 2.08 | 1.34 | 1.49 | 1.92 |
| MR7 | 1.40 | 1.24 | 1.00 | .71 | .64 | .80 |
| MR8 | 1.98 | 2.12 | 1.89 | 1.02 | 1.00 | .95 |
| MR9 | 4.53 | 4.41 | 3.48 | 2.70 | 2.52 | 2.90 |
| MR10 | 1.36 | 1.09 | 1.16 | .77 | .70 | .77 |
| MR11 | 3.26 | 3.09 | 2.09 | 1.64 | 1.97 | 2.14 |
| MR12 | 6.95 | 7.13 | 6.11 | 4.70 | 5.39 | 5.82 |
| | | | | | | |
| TG1 | .28 | .26 | .45 | .48 | .21 | .42 |
| TG2 | .21 | .18 | .52 | .39 | .16 | .38 |
| TG3 | .60 | .56 | 1.03 | .86 | .48 | .75 |
| TG4 | .52 | .49 | .94 | .80 | .41 | .79 |
| TG5 | .93 | .96 | 1.87 | 1.56 | .73 | 1.23 |
| TG6 | .73 | .70 | 1.51 | 1.21 | .58 | 1.38 |
| TG7 | 1.63 | 1.62 | 2.74 | 2.22 | .96 | 1.91 |
| TG8 | 1.56 | 1.50 | 2.50 | 1.99 | 1.12 | 2.02 |
| | | | | | | |
| H1 | .07 | .06 | .10 | .11 | .06 | .05 |
| H2 | .57 | .52 | .68 | .65 | .43 | .42 |
| H3 | 1.82 | 1.70 | 2.40 | 2.35 | 1.39 | 1.35 |
| H4 | 4.22 | 3.98 | 4.35 | 4.43 | 3.22 | 3.09 |
| H5 | 8.12 | 7.67 | 8.91 | 8.66 | 6.16 | 5.92 |

## Choice of Start Procedure

Four basic start procedures (STEP 1) were implemented. The first code, referred to as LIFO, uses the last-in first-out or depth first rule to construct the initial basis tree. The second code, FIFO, uses the similar first-in first-out rule. Both of these procedures require the arcs to be processed in forward star sequence. The balanced tree start was implemented in the code BAL and the modified balanced tree start was implemented in MODBAL. The MODBAL implementation uses the parameter settings $n_1 = .10|N|$ and $n_2 = .75|N|$ which cause the final pass to begin as soon as an initial pass fails to add at least 10% of the nodes to the tree or as soon as the tree contains at least 75% of the network nodes. Limited testing with other parameter settings indicated that these values were quite robust across all problem topologies.

All four implementations used the same entering arc selection rule. Specifically, a candidate list of length twenty was used. The SEQ/CS criteria was used to load the list and the BEP/NS criteria was used to select entering arcs from the list.

The testing indicated that the LIFO and FIFO codes perform approximately the same in terms of solution time. The starting bases generated by the two codes, however, are quite different. As expected, the LIFO code constructs a thin, deep initial basis tree and the FIFO code constructs a fat, shallow initial basis tree. However, the structure of the optimal basis tree does not resemble that of either initial tree.

The principal shortcoming of both codes is that they initially place all nodes except the terminal on the s-side of the tree. This makes the identification of eligible entering arcs somewhat difficult during the initial pivots, as reflected by the fact that the average number of arcs examined per pivot is much higher during the early pivots than during the middle pivots for both the LIFO and FIFO code.

Computational tests comparing the modified balanced tree start MODBAL and the standard balanced tree start BAL indicate MODBAL is clearly superior. This superiority is most pronounced on the multi-terminal random problem class. On selected problems in this class, MODBAL outperformed BAL on all but one problem.

Due to the similarity of the LIFO and FIFO times, and the superiority of MODBAL over BAL times, only the results for LIFO and MODBAL are presented in Table V. These results indicate that the modified balanced tree start is superior to the other start procedures tested, particularly for the problem classes designed to simulate multi-terminal networks (i.e., multi-terminal random and transit grid).

## Choice of Entering Arc

Although many codes were developed to test the impact of the entering arc selection rule (STEP 3), only six of the codes will be presented in any detail. All codes use the modified balanced tree start procedure, the first minimum rule for selecting the leaving arc, and the forward star form for storing the original problem data.

The first three codes, SEQNS, SEQCS, and SEQPS, were developed to test the impact of suboptimization on overall solution speeds. SEQNS uses the sequential pivot selection rule with no suboptimization. SEQCS, on the other hand, uses the same sequential entering arc criterion, but requires complete suboptimization of the problem using only case (a) entering arcs before any case (b) entering arcs are allowed. SEQPS uses the sequential criterion with partial suboptimization. Phase I (case (a) pivots only) was terminated after $p = .50|N|$ pivots.

Computational testing indicated that SEQCS is superior to SEQNS on all but the smallest networks. The superiority of SEQCS is particularly evident on the large multi-terminal random problems where SEQNS ran as much as 60% slower than SEQCS.

The performance of SEQPS is harder to evaluate. Neither SEQPS nor SEQCS dominated the other. However, SEQCS is highly robust, yielding good solution times for all problems, while SEQPS yields solution times whose quality is far more variable. Barring the possibility of finding a value for the p parameter for SEQPS that improves its stability, the SEQCS code is preferable for situations in which robustness is valued.

The next three codes tested use a candidate list of length twenty to control the selection of entering arc. Each employs the same criterion for reloading and redimensioning the list: reloading occurs when all pivot eligible arcs in the list have been used; redimensioning occurs when a complete pass of the arc data fails to yield enough arcs to fill the list (whereupon the dimension of the list is set equal to the number of pivot eligible arcs actually found).

All three codes use the SEQ/CS criterion to load the candidate list. The first code, referred to as CANSEQ, uses the SEQ/NS criterion to select the entering arc from the candidate list, while the second code, CANBEP, uses the BEP/NS criterion, and the third code, CANAUG, uses the AUG/NS criterion.

CANAUG turned out to be a definite loser. The reason for its poor performance is that for each pivot, the basis equivalent path of every arc in the candidate list must be traversed in order to identify the arc with the maximum minimum ratio. CANAUG tends to require fewer pivots than the other codes tested, but its solution times ranged from 50% to 300% slower. Other implementations of the largest augmentation criteria, including candidate list and non-candidate list codes with no and partial suboptimization, performed just as badly.

Table V presents the solution times for SEQCS and CANBEP. In the table, CANBEP is referred to as MODBAL since it is the same code used to test the choice of starting bases. The performance of these two codes is basically the same on the random and multi-terminal random problems, but SEQCS dominates CANBEP on the transit grid and hard problems. SEQCS runs up to twice as fast as CANBEP on the transit grid problems.

A partial explanation of the poor performance of the minimum basis equivalent path length criterion on the transit grid problems is that it generates more pivots, both total and degenerate, than the sequential criterion. For the transit grid problems, the lengths of the flow augmenting paths (non-degenerate basis equivalent paths) tend to be long. By concentrating on the short basis equivalent paths, CANBEP ends up doing more work than the approach used in SEQCS.

## Choice of Data Structure

The last question considered regards the choice of data structure for storing the original problem data. All codes previously discussed made use of the forward star form. The best such code appears to be SEQCS. A number of codes using the random form were also developed. The best of these, simply referred to as RANDOM, uses the modified balanced tree start procedure, the SEQ/NS entering arc selection rule, and the first minimum found leaving arc selection rule. Table V presents the times for the best forward star form code, SEQCS, and the best random form code, RANDOM. The results are fairly clear: SEQCS is faster than RANDOM for almost all problems, except notably the hard problems. The most pronounced superiority of SEQCS is for the transit grid problems which have a large number of arcs incident on the source and terminal nodes (problem sets TG2, TG4, TG6, and TG8). On these problems SEQCS runs as much as twice as fast as RANDOM.

The overall conclusion from this computational testing of the primal simplex maximum flow codes is that SEQCS is the most consistent winner. To recapitulate, SEQCS uses the modified balanced tree start (with $n_1 = .10|N|$ and $n_2 = .75|N|$), the sequential entering arc selection rule with complete suboptimization, the first minimum rule for selecting the leaving arc, and the forward star form for storing the original problem data.

## 5.0 LABEL TREE METHODS

### 5.1 Algorithm Features

All label tree methods may be viewed as variants of the original Ford and Fulkerson algorithm [9, 15, 16, 17, 18]. These methods alternate between a *node labeling* step and a *flow augmentation* step in which the flow from s to t is increased. The methods terminate when the terminal node cannot be labeled in a sequence of label assignments starting from the source.

The purpose of the labeling step is to identify a *flow augmentation path*, which consists of a set of arcs connecting labeled nodes along which additional flow may be transmitted from s to t. The *allowable flow change*, $\delta$, is the value of the maximum flow change which can be accommodated by all arcs of this path while keeping their flows within bounds (hence maintaining a feasible solution to (1)-(6)). During the flow augmentation step, the flows on each arc in this path are adjusted by $\delta$, i.e., are increased by $\delta$ if the arc's direction coincides with the path orientation, and are decreased by $\delta$ otherwise.

Generally speaking, the labeling step consists of scanning labeled nodes and attaching labels to label eligible nodes. Node j is said to be *label eligible* from node i if j is currently unlabeled and the flow on arc (i,j) can be increased or the flow on arc (j,i) decreased. The associated arc is referred to as *flow eligible*. Node i is *scanned* by sequentially examining each of its incident arcs (into or out of node i) and labeling the label eligible nodes. Thus, the labeling step creates a tree structure, henceforth referred to as a *label tree* (or *flow augmentation tree*).

The labeling step terminates when the terminal is labeled since a flow augmentation path has been determined. Consequently, while the label tree resembles the basis tree of Section 4, it usually contains only a proper subset of the nodes and is therefore often much smaller.

The basic steps of label tree methods are as follows:

STEP 1: [INITIALIZATION] Label node s and create a label tree $T(N_L,A_L)$ such that $N_L = \{s\}$ and $A_L = \emptyset$. Set $L = \{s\}$.

STEP 2: [NODE SELECTION] If $L = \emptyset$, go to STEP 6. Otherwise, select some node i from the set L of labeled and unscanned nodes and delete i from L.

STEP 3: [SCAN NODE] Scan node i assigning labels to all label eligible nodes and add these nodes and arcs to the label tree $T(N_L,A_L)$ and the newly labeled nodes to L. If node t is unlabeled, go to STEP 2.

STEP 4: [FLOW AUGMENTATION] Determine the allowable flow change $\delta$ on the unique path from s to t in the label tree and adjust the flows on the arcs of this path by $\delta$.

STEP 5: [UPDATE LABEL TREE] Erase or update the label tree. If the label tree is erased, go to STEP 1. Otherwise, adjust the label tree and the nodes in L appropriately and go to STEP 2.

STEP 6: [TERMINATE] Stop. A set of maximum flow values has been determined.

To facilitate the discussion of alternative implementations of label tree methods, we define the *active star* of node i to be the arcs of i's complete star whose flows are capable of being adjusted by a positive $\delta$ value (i.e., increased by $\delta$ if the arc is directed out of node i, and decreased by $\delta$ if the arc is directed into node i). As before, we also refer to the endpoints of arcs in a star (complete, active, etc.) for node i as elements of the star--excepting node i itself. Thus, in particular, the arcs (nodes) in i's active star are precisely those that would be determined flow eligible (label eligible) when scanning node i if none of these elements were in the current label tree.

We envisioned the primary strategic possibilities for creating an efficient implementation (version) of the label tree methods to be embedded in the following considerations:

(a) the use of a partition scheme that isolates the active star of each node (to expedite the performance of STEP 3).

(b) the rule used in STEP 2 to select the next labeled node to scan.

(c) the portion of the label tree re-used in STEP 5.

(d) the contents of the label assigned in STEP 3 to a node in the label tree.

Consideration (a) does not affect the composition of the label tree, but does affect its speed of creation. In particular, this consideration addresses the fact that scanning a labeled node is a time-consuming process and depends importantly on the way in which problem data are stored and accessed. Considerations (b) and (c) relate to the size, shape, and the re-use of the label tree. The node label contents of consideration (d) consist of one or two parts. One part only affects tree traversal and label re-use. The other part, if present, determines the order in which the labeled nodes are scanned.

The next subsection describes the alternatives explored in this study for storing and accessing problem data, and their relevance to consideration (a). Later subsections then address considerations (b)-(d).

## 5.2 Problem Data Storage, Labeling, and Partitioning

The first data structure used for storing the problem is called the *modified forward star* (*MFS*) form. This requires three $|A|$ length lists, TO, XOUT, and XIN, to represent the original problem and its current solution. The MFS form stores all arcs of a given forward star in consecutive locations, and further stores the forward star for node i just before the forward star of node i + 1. As a result, it is unnecessary to explicitly store the from node of each arc. Instead, the list of to nodes, TO, is flagged (a simple negation) to indicate the beginning of a new node's forward star. By eliminating the list of from nodes, this data structure effectively restricts the algorithm to sequential processing of the arc data.

In addition to the to node, the MFS form stores the allowable flow increase, XOUT, and decrease, XIN, for each arc. Although equivalent to the arc capacity and flow, these lists streamline the execution of STEP 3 of the basic labeling algorithm. Figure 6 illustrates the MFS form for a feasible solution to the problem given in Figure 1.

Since this data structure restricts the algorithm to sequential processing of the arc data (and hence sequential scanning of the nodes), STEPS 2 and 3 of the basic algorithm must be slightly modified. Specifically, instead of examining the arcs entering and leaving the labeled nodes, the algorithm must examine the arcs leaving the labeled and unlabeled nodes. This means that arc (i,j) can be used to label node j (i) if node i is labeled (unlabeled) and node j is unlabeled (labeled).

The second data structure for storing the problem is called the *forward star linked reverse star* (*FSLRS*) form. It represents a generalization of the MFS form. The FSLRS form requires that the arcs are stored in the same sequence (sorted by from nodes, in consecutive order) as the MFS form. Unlike the MFS form, the FSLRS form allows the algorithm to process the arcs in virtually any order. This added capability requires the use of two additional $|A|$ length lists, FROM and LINK, and two $|N|$ length lists, OUT and IN. The FROM list stores the from node of each arc, and the LINK list serves as a reverse star linkage for the arcs. The OUT (IN)

FIGURE 6

MODIFIED FORWARD STAR FORM

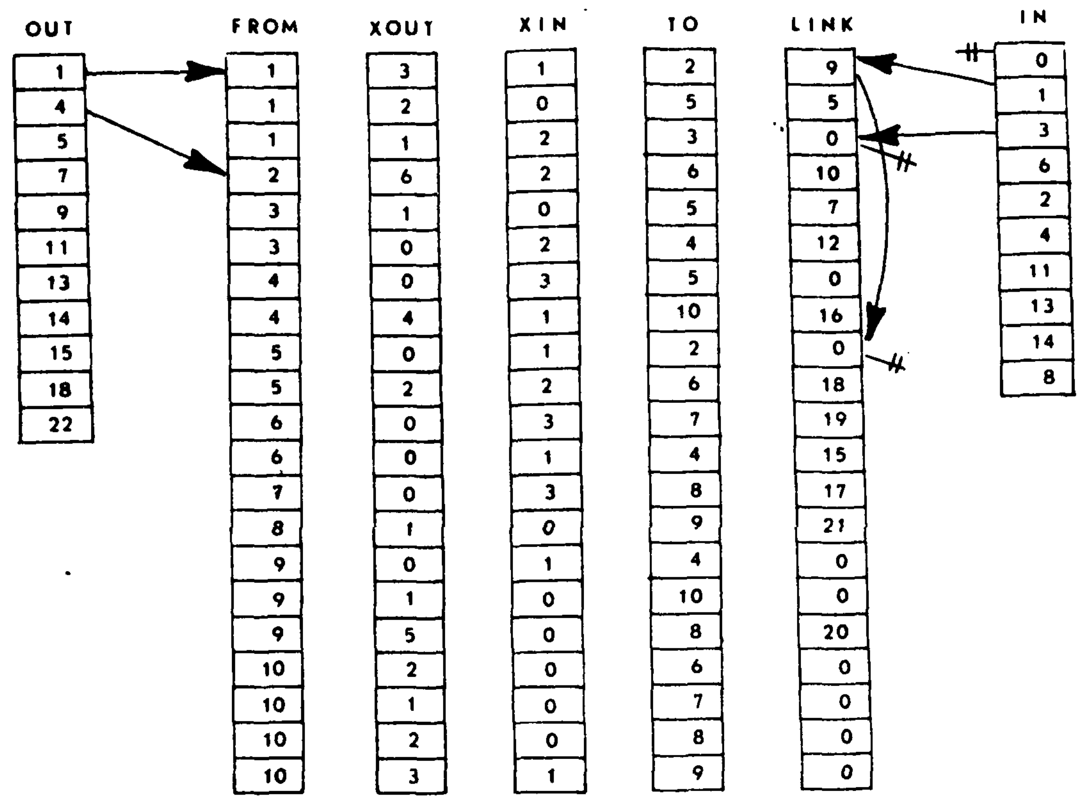
| ARC | TO | XOUT | XIN |
|---|---|---|---|
| 1 | -2 | 3 | 1 |
| 2 | 5 | 2 | 0 |
| 3 | 3 | 1 | 2 |
| 4 | -6 | 6 | 2 |
| 5 | -5 | 1 | 0 |
| 6 | 4 | 0 | 2 |
| 7 | -5 | 0 | 3 |
| 8 | 10 | 4 | 1 |
| 9 | -2 | 0 | 1 |
| 10 | 6 | 2 | 2 |
| 11 | -7 | 0 | 3 |
| 12 | 4 | 0 | 1 |
| 13 | -8 | 0 | 3 |
| 14 | -9 | 1 | 0 |
| 15 | -4 | 0 | 1 |
| 16 | 10 | 1 | 0 |
| 17 | 8 | 5 | 0 |
| 18 | -6 | 2 | 0 |
| 19 | 7 | 1 | 0 |
| 20 | 8 | 2 | 0 |
| 21 | 9 | 3 | 1 |
| 22 | 0 | | |

list points to the first arc in a node's forward (reverse) star. Figure 7 illustrates the FSLRS form for the example problem. This data structure, although requiring virtually twice the core of the MFS form, enables the algorithm to scan the nodes (STEPS 2 and 3) in any order whereas the MFS form is restricted to sequential scanning. This added flexibility is used to improve the over-all solution characteristics of the labeling algorithm.

The MFS and FSLRS forms can be implemented using four $|N|$ length lists to represent the label tree. Two of these lists, the predecessor

FIGURE 7

FORWARD STAR LINKED REVERSE STAR FORM



| OUT | FROM | XOUT | XIN | TO | LINK | IN |
|---|---|---|---|---|---|---|
| 1 | 1 | 3 | 1 | 2 | 9 | 0 |
| 4 | 1 | 2 | 0 | 5 | 5 | 1 |
| 5 | 1 | 1 | 2 | 3 | 0 | 3 |
| 7 | 2 | 6 | 2 | 6 | 10 | 6 |
| 9 | 3 | 1 | 0 | 5 | 7 | 2 |
| 11 | 3 | 0 | 2 | 4 | 12 | 4 |
| 13 | 4 | 0 | 3 | 5 | 0 | 11 |
| 14 | 4 | 4 | 1 | 10 | 16 | 13 |
| 15 | 5 | 0 | 1 | 2 | 0 | 14 |
| 18 | 5 | 2 | 2 | 6 | 18 | 8 |
| 22 | 6 | 0 | 3 | 7 | 19 | |
| | 6 | 0 | 1 | 4 | 15 | |
| | 7 | 0 | 3 | 8 | 17 | |
| | 8 | 1 | 0 | 9 | 21 | |
| | 9 | 0 | 1 | 4 | 0 | |
| | 9 | 1 | 0 | 10 | 0 | |
| | 9 | 5 | 0 | 8 | 20 | |
| | 10 | 2 | 0 | 6 | 0 | |
| | 10 | 1 | 0 | 7 | 0 | |
| | 10 | 2 | 0 | 8 | 0 | |
| | 10 | 3 | 1 | 9 | 0 | |

node and predecessor arc are used in STEP 4 to augment the flow along the flow augmentation path from s to t. The other two lists, the thread and depth functions, are used in STEP 5 to update the label tree after flow augmentation. These last two lists are not used if the "update" consists of a complete erasure of the tree.

It is also possible to implement the FSLRS form using an additional $|N|$ length list, L, to control the selection in STEP 2 of the nodes to scan. Depending upon the implementation, the L list may be maintained as a FIFO, LIFO, largest augmentation, or minimum depth sequence list.

The third data structure stores the problem in a *fixed* mirror arc form, henceforth called the FIXMIR form. (This set of data structures was developed in [4a].) To understand this form, note that it must be possible to access the data for an arc (u,v) from either of its two endpoints, depending on whether the flow is to be increased, (scanning from u) or decreased (scanning from v). Moreover, when scanning from u it is only necessary to know the arc's net capacity in the forward direction (capacity minus current flow) and when scanning from v it is only necessary to know the net capacity in the reverse direction (current flow). In each case, it is necessary to identify the endpoint of the arc opposite from the node being scanned.

Thus, in particular, the relevant information for arc (u,v) is conveniently subdivided and stored in two places. If this arc is assigned a numerical index k (by which we may then refer to (u,v) as the $k^{th}$ arc), then we may store the "opposite node" and net capacity values in the $k^{th}$ and $|A| + k^{th}$ position of two arrays OPNODE and NETCAP. For the example network, arc (2,6) may be considered to be the $4^{th}$ arc out of 21, with a capacity of 8. If, at some stage of an algorithm, the arc's current flow is 5, then OPNODE(4) = 6, OPNODE(25) = 2, NETCAP(4) = 3, and NETCAP(25) = 5. Note that the original capacity of an arc is always equal to the sum of its two stored NETCAP values, and so does not have to be stored separately.

Having recorded the information for an arc (u,v) in two different places, we may just as easily regard this information as applying to *two different arcs*, one from node u to node v, and the other from node v to node u. Thus the OPNODE value for each of these two arcs may be viewed as naming the *to node*. These arcs are, of course, intimately related in that the OPNODE of one is actually the *from node* of the other. Moreover, any change in the NETCAP value of one must be accompanied by a change equal in magnitude and opposite in direction in the NETCAP value of the other. Because of these symmetric relationships, it is unnecessary to distinguish which of these two arcs corresponds to an original arc of the problem (except for purposes of identifying the final solution), and each arc is called the *mirror* of the other. Henceforth, in the FIXMIR data structure, each of the two mirror instances of an original arc will be viewed as a unique arc in its own right, and an *arc index* will be referred to as the index under which such an arc's information is stored in the OPNODE and NETCAP arrays.

To make the symmetry complete in all aspects, instead of recording information about the two mirror instances of arc k in positions k and A + k of OPNODE and NETCAP, this information could be recorded in positions k and M - k where $M = 2|A| + 1$. Then if h denotes either of these two positions, the other position is M - h. This schema was tested but rejected on two counts. The implementations of this study can be utilized in both a subroutine setting and a "stand-alone" fashion. With respect to subroutine situations, it is important to note that, by using positions k and $|A| + k$ to store arc data, the OPNODE array contains all of the from nodes of the original arcs followed by all of the to nodes. So, the OPNODE array may be instantly created by placing the from node array immediately after the to node array in a common block. If the arc data are stored in positions k and M - k, then the order of the entries in the user's from node array must be reversed upon entering and leaving the maximum flow subroutine. Further, our testing indicated that the next data structure to be discussed is more suitable for stand-alone applications.

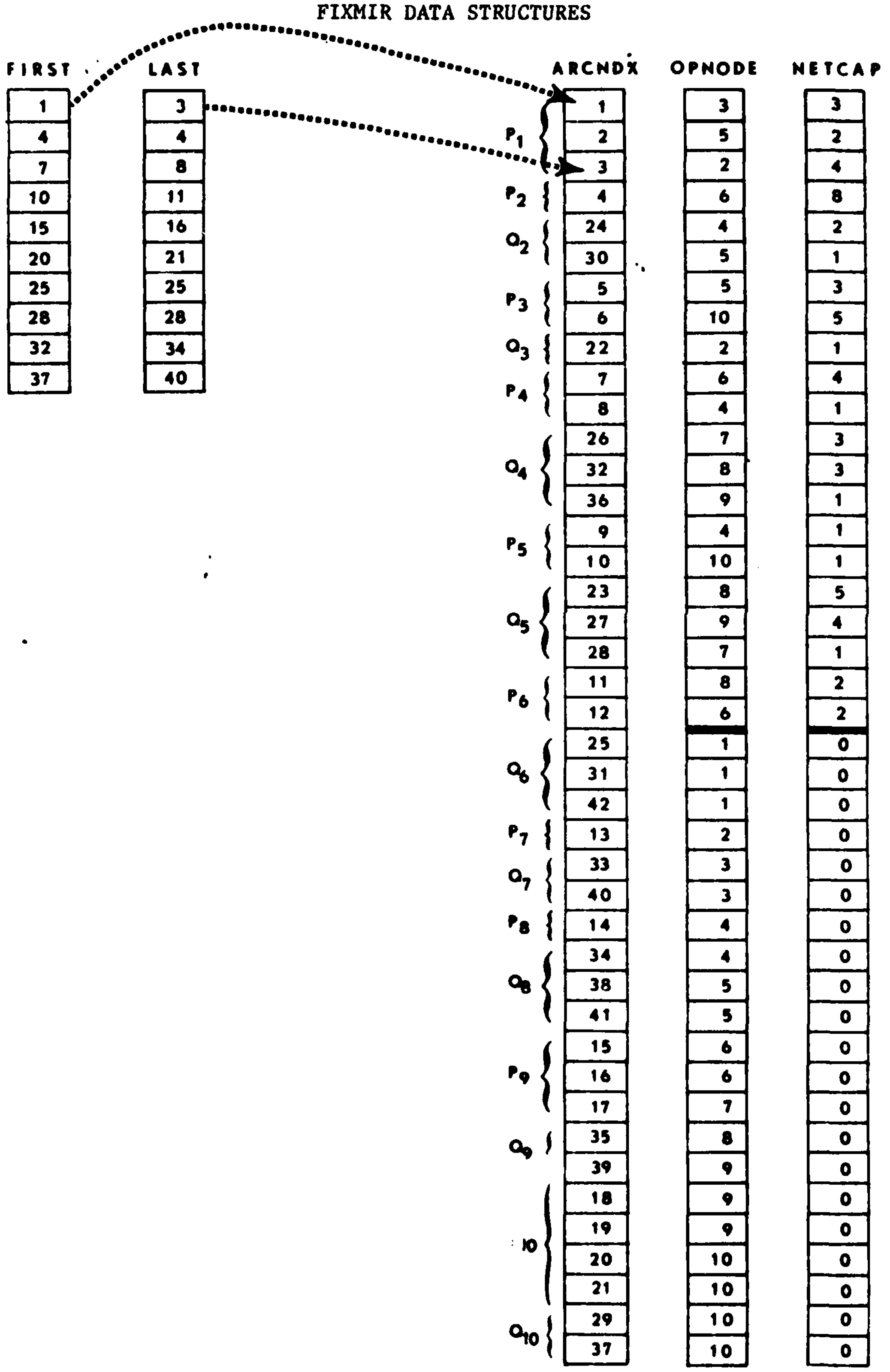
To make use of the OPNODE and NETCAP arrays it is desirable to be able to access all arc indexes associated with any given node (in the role of *from node*) in a convenient fashion. This may be done by storing all arc indexes for such a node sequentially in an array ARCNDX(I), I = FIRST(NODE) to FIRST(NODE + 1) - 1. Thus, for example, from Figure 1, node 2 is an endpoint of the original arcs (2,6), (1,2), and (5,2) which may be respectively indexed 4, 3, and 9 (out of 21 arcs). Then in FIXMIR structure node 2 takes the role of *from node* for the arcs (2,6), (2,1), and (2,5) which are respectively indexed 4, 24 (= 21 + 3), and 30 (= 21 + 9). Furthermore, if the first position I in ARCNDX for storing node 2's arc indexes is I = 4, then FIRST(2) = 4, ARCNDX(4) = 4, ARCNDX(5) = 24, ARCNDX(6) = 30. The arc indexes associated with node 3 would thus begin in the next position I = 7 of the ARCNDX array (identifying the last position for the arcs associated with node 2 as FIRST(2 + 1) - 1). Since the arc indexes of ARCNDX(I) for I = FIRST(NODE) to FIRST(NODE + 1) - 1 name *all* arcs associated with NODE, this array actually records the complete star of NODE. (However, in the FIXMIR structure, the complete star for any given node is actually treated as a forward star, since the node is treated as *from node*.) Figure 8 contains the FIXMIR data structures for the example network.

Using this data structure, the node scan of node i can be carried out by checking NETCAP to determine an arc's flow eligibility and then checking the label status of the appropriate to nodes. However, this procedure may be further streamlined by partitioning the arc indexes recorded in ARCNDX(I), I = FIRST(NODE) to FIRST(NODE + 1) - 1 into two consecutive groups in accordance with consideration (a), the first group naming the arcs with positive net capacity (hence in the active star) and the second group naming those with zero net capacity. This may be done by introducing the array LAST(NODE), where all arcs in NODE's active star are named in ARCNDX(I) for I = FIRST(NODE) to LAST(NODE). The likelihood of computational savings by identifying the active forward star in this partitioning scheme is apparent since the amount of effort to "reconstitute" the partition is minimal and occurs only during the flow augmentation phase.

In summary, several advantages are provided by the FIXMIR structure. First, while additional arrays are introduced, the original arc data is unaltered and easily retrievable in its original order. Second, the node scan procedure is greatly simplified and accelerated. Third, the structure readily admits the use of a partition which further simplifies the scanning procedure and reduces the number of arcs examined.

FIGURE 8

FIXMIR DATA STRUCTURES

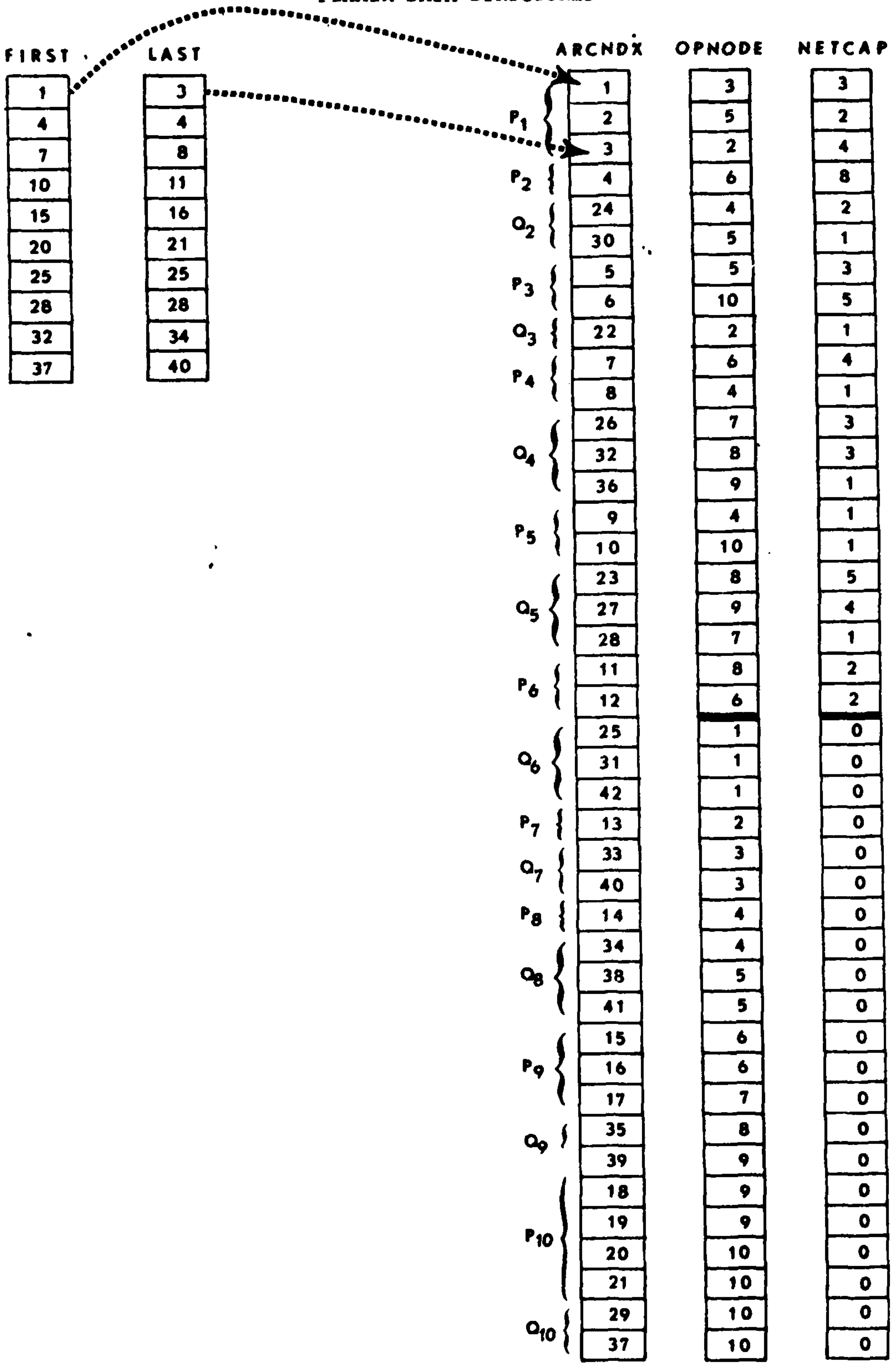


| FIRST | LAST |
|---|---|
| 1 | 3 |
| 4 | 4 |
| 7 | 8 |
| 10 | 11 |
| 15 | 16 |
| 20 | 21 |
| 25 | 25 |
| 28 | 28 |
| 32 | 34 |
| 37 | 40 |

| | ARCNDX | OPNODE | NETCAP |
|---|---|---|---|
| $P_1$ | 1 | 3 | 3 |
| | 2 | 5 | 2 |
| | 3 | 2 | 4 |
| $P_2$ | 4 | 6 | 8 |
| $Q_2$ | 24 | 4 | 2 |
| | 30 | 5 | 1 |
| $P_3$ | 5 | 5 | 3 |
| | 6 | 10 | 5 |
| $Q_3$ | 22 | 2 | 1 |
| $P_4$ | 7 | 6 | 4 |
| | 8 | 4 | 1 |
| $Q_4$ | 26 | 7 | 3 |
| | 32 | 8 | 3 |
| | 36 | 9 | 1 |
| $P_5$ | 9 | 4 | 1 |
| | 10 | 10 | 1 |
| $Q_5$ | 23 | 8 | 5 |
| | 27 | 9 | 4 |
| | 28 | 7 | 1 |
| $P_6$ | 11 | 8 | 2 |
| | 12 | 6 | 2 |
| $Q_6$ | 25 | 1 | 0 |
| | 31 | 1 | 0 |
| | 42 | 1 | 0 |
| $P_7$ | 13 | 2 | 0 |
| $Q_7$ | 33 | 3 | 0 |
| | 40 | 3 | 0 |
| $P_8$ | 14 | 4 | 0 |
| $Q_8$ | 34 | 4 | 0 |
| | 38 | 5 | 0 |
| | 41 | 5 | 0 |
| $P_9$ | 15 | 6 | 0 |
| | 16 | 6 | 0 |
| | 17 | 7 | 0 |
| $Q_9$ | 35 | 8 | 0 |
| | 39 | 9 | 0 |
| 10 | 18 | 9 | 0 |
| | 19 | 9 | 0 |
| | 20 | 10 | 0 |
| | 21 | 10 | 0 |
| $Q_{10}$ | 29 | 10 | 0 |
| | 37 | 10 | 0 |

FIGURE 8

FIXMIR DATA STRUCTURES



| FIRST | LAST |
|---|---|
| 1 | 3 |
| 4 | 4 |
| 7 | 8 |
| 10 | 11 |
| 15 | 16 |
| 20 | 21 |
| 25 | 25 |
| 28 | 28 |
| 32 | 34 |
| 37 | 40 |

| | ARCNDX | OPNODE | NETCAP |
|---|---|---|---|
| $P_1$ | 1 | 3 | 3 |
| | 2 | 5 | 2 |
| | 3 | 2 | 4 |
| $P_2$ | 4 | 6 | 8 |
| $Q_2$ | 24 | 4 | 2 |
| | 30 | 5 | 1 |
| $P_3$ | 5 | 5 | 3 |
| | 6 | 10 | 5 |
| $Q_3$ | 22 | 2 | 1 |
| $P_4$ | 7 | 6 | 4 |
| | 8 | 4 | 1 |
| $Q_4$ | 26 | 7 | 3 |
| | 32 | 8 | 3 |
| | 36 | 9 | 1 |
| $P_5$ | 9 | 4 | 1 |
| | 10 | 10 | 1 |
| $Q_5$ | 23 | 8 | 5 |
| | 27 | 9 | 4 |
| | 28 | 7 | 1 |
| $P_6$ | 11 | 8 | 2 |
| | 12 | 6 | 2 |
| $Q_6$ | 25 | 1 | 0 |
| | 31 | 1 | 0 |
| | 42 | 1 | 0 |
| $P_7$ | 13 | 2 | 0 |
| $Q_7$ | 33 | 3 | 0 |
| | 40 | 3 | 0 |
| $P_8$ | 14 | 4 | 0 |
| $Q_8$ | 34 | 4 | 0 |
| | 38 | 5 | 0 |
| | 41 | 5 | 0 |
| $P_9$ | 15 | 6 | 0 |
| | 16 | 6 | 0 |
| | 17 | 7 | 0 |
| $Q_9$ | 35 | 8 | 0 |
| | 39 | 9 | 0 |
| $P_{10}$ | 18 | 9 | 0 |
| | 19 | 9 | 0 |
| | 20 | 10 | 0 |
| | 21 | 10 | 0 |
| $Q_{10}$ | 29 | 10 | 0 |
| | 37 | 10 | 0 |

The major disadvantage of the FIXMIR structure is the necessity to access the ARCNDX array before retrieving information from the OPNODE and NETCAP arrays. This shortcoming led us to consider a fourth data organization which eliminates the intermediate ARCNDX array.

The efficiency of the FIXMIR data structures depends on (1) the storage format of the arc data and (2) the grouping of the arc indexes into complete stars and active stars. It is desirable to maintain these two properties but to eliminate the intermediate access array.

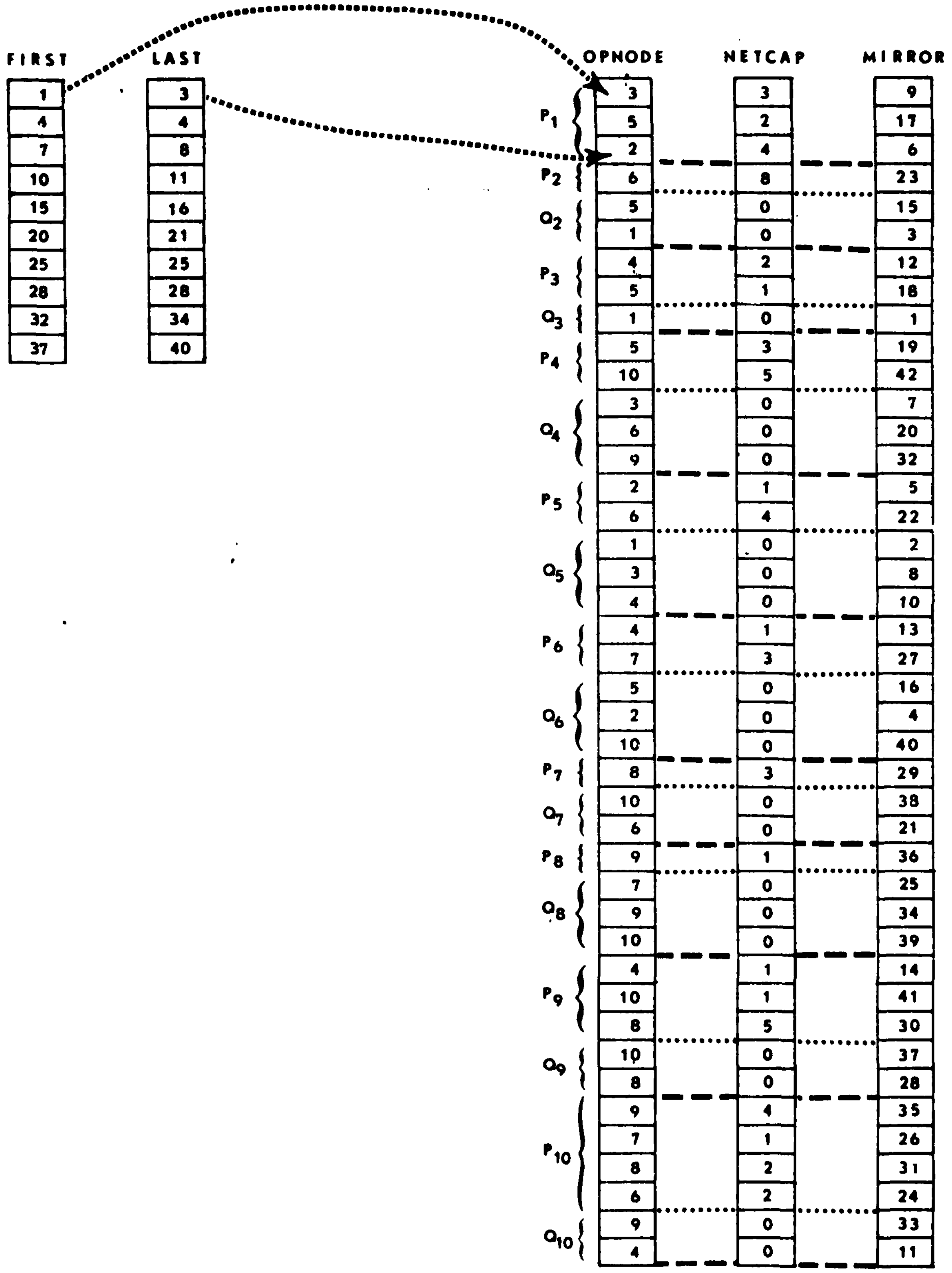
These objectives may be satisfied by the dynamic mirror, or DYNMIR structure, which arranges the arc data in the same order previously indicated by the ARCNDX entries. The FIRST and LAST arrays point directly into the arc data (if a partition is used). By ordering the data in this fashion, it is no longer possible to maintain the entries of the OPNODE and NETCAP arrays in fixed positions because these entries must be moved each time an active star is altered. Further, when these entries are moved, it may no longer be possible to find an arc's mirror in its original position. Thus, for a given arc currently referenced by the arc index k, the current index of its mirror is recorded in MIRROR(k). The dimension of MIRROR, as the dimension of OPNODE and NETCAP, must be twice the number of original arcs. Figure 9 presents the entries of the DYNMIR arrays for the example problem.

Other than requiring only a single access to obtain individual arc data components, the DYNMIR structure executes STEP 3 in the exact fashion of the FIXMIR structure. The partition update of the active star for this structure is somewhat more complicated: entries in both the OPNODE and NETCAP arrays are swapped while some MIRROR entries need to be modified accordingly.

While the DYNMIR structure allows highly efficient processing, its practicality is rather limited. It is not suitable for situations which demand that the final arc flows be output in the same sequence in which the arcs were originally input. The additional computational time and storage required to recover the original sequence order overcomes the advantages otherwise gained by improved speed of execution during the internal solution phase. Rather, the DYNMIR structure finds its usefulness in those situa-

FIGURE 9

DYNMIR DATA STRUCTURES



| FIRST | LAST |
|---|---|
| 1 | 3 |
| 4 | 4 |
| 7 | 8 |
| 10 | 11 |
| 15 | 16 |
| 20 | 21 |
| 25 | 25 |
| 28 | 28 |
| 32 | 34 |
| 37 | 40 |

| | OPNODE | NETCAP | MIRROR |
|---|---|---|---|
| $P_1$ | 3 | 3 | 9 |
| | 5 | 2 | 17 |
| | 2 | 4 | 6 |
| $P_2$ | 6 | 8 | 23 |
| $Q_2$ | 5 | 0 | 15 |
| | 1 | 0 | 3 |
| $P_3$ | 4 | 2 | 12 |
| | 5 | 1 | 18 |
| $Q_3$ | 1 | 0 | 1 |
| $P_4$ | 5 | 3 | 19 |
| | 10 | 5 | 42 |
| $Q_4$ | 3 | 0 | 7 |
| | 6 | 0 | 20 |
| | 9 | 0 | 32 |
| $P_5$ | 2 | 1 | 5 |
| | 6 | 4 | 22 |
| $Q_5$ | 1 | 0 | 2 |
| | 3 | 0 | 8 |
| | 4 | 0 | 10 |
| $P_6$ | 4 | 1 | 13 |
| | 7 | 3 | 27 |
| $Q_6$ | 5 | 0 | 16 |
| | 2 | 0 | 4 |
| | 10 | 0 | 40 |
| $P_7$ | 8 | 3 | 29 |
| $Q_7$ | 10 | 0 | 38 |
| | 6 | 0 | 21 |
| $P_8$ | 9 | 1 | 36 |
| $Q_8$ | 7 | 0 | 25 |
| | 9 | 0 | 34 |
| | 10 | 0 | 39 |
| $P_9$ | 4 | 1 | 14 |
| | 10 | 1 | 41 |
| | 8 | 5 | 30 |
| $Q_9$ | 10 | 0 | 37 |
| | 8 | 0 | 28 |
| $P_{10}$ | 9 | 4 | 35 |
| | 7 | 1 | 26 |
| | 8 | 2 | 31 |
| | 6 | 2 | 24 |
| $Q_{10}$ | 9 | 0 | 33 |
| | 4 | 0 | 11 |

tions where it is adequate to identify an optimal solution simply by naming arc endpoints and flow values or where the network is initially organized in the complete star arc format (and the sequence of arcs within any complete star is immaterial).

Despite the limited usefulness of the DYNMIR structure in applications, the evaluation of its performance serves two purposes. First, it provides a comparison with the implementations of Bayer [5] and Cheung [6] which require that the problem network be in complete star format. Second, because this structure is more efficient than the others described, it provides an indicator of the best possible performances that can be obtained via experimenting with considerations (b)-(d).

## 5.3 Label Tree Procedure

In treating consideration (d), the labeling schemes of earlier implementations were greatly invluenced by the work of Ford and Fulkerson [9, 15, 16, 17, 18], who proposed a two-part label. The first part contains the index of the path predecessor of the labeled node. Thus, if node j is labeled when scanning node i (via an original arc of the form (i,j) or (j,i)), then the index i is stored as the first part of the label assigned to node j. Node i and arc (i,j) (or (j,i)), are referred to as the *predecessor node* and *arc*, respectively, of node j. (This predecessor terminology is consistent with that employed in discussing the primal algorithm.) The second part of node j's label contains the maximum allowable flow change on the labeled path from the source to node j. As soon as the terminal node is labeled, the algorithm accesses the second part of the terminal node label to obtain the flow change to be made on the flow augmentation path in executing STEP 4.

By contrast, most of the following implementations utilize only the first part of the two-part label (the predecessor node index), and make two additional passes of the arcs in the flow augmentation path (rather than one) when the terminal node is labeled. The first pass computes $\delta$ while the second performs the flow updates. This modification reduces the computational effort because the flow augmentation paths of most label trees contain only a few of the arcs in these trees. Further, if the partitioning scheme identifying active stars in either the FIXMIR or DYNMIR data organizations is used, then the resulting streamlined node scan makes the use of a single ("one-part") label even more attractive.

Another still more straightforward modification of the label contents is employed in many of our label tree implementations: the predecessor *node* index is instead replaced by a predecessor *arc* index which is stored in a node length array, PREARC. The reason for using PREARC is that the standard prescription of maintaining predecessor nodes allows direct accessing of arc data only if the arc data is stored in *matrix* form, that is, where arc (i,j) is accessed via nodes i and j. However, since each of our four data organizations store the arc data sequentially, it is more reasonable to store the predecessor arc number. This avoids a search each time information other than the identity of an arc's endpoints is required.

### 5.3.1 FIFO

The simplest methods for implementing the labeling step determine the sequence in which the labeled nodes are selected for scanning by the sequence in which they are labeled. The first of these methods utilizes a node length array, NEXNOD ("next node"), to store labeled nodes sequentially and then scans them in that order. Commonly referred to as a *First-In First-Out* or *FIFO* node scan, this method adds nodes to the label tree level by level. Thus, the label tree is created in *breadth first* fashion.

### 5.3.2 LIFO

Another common method stores the nodes sequentially but scans the last rather than the first labeled but unscanned node. The sequential list NEXNOD of labeled and unscanned nodes is processed as a push down stack removing nodes from the end rather than the beginning. This scanning order is commonly described as *Last-In First-Out* or *LIFO*.

We considered two versions of the LIFO procedure. The first, called simply the LIFO scan, examines the arcs out of a labeled node only until a label eligible arc is encountered. Then the opposite endpoint of this arc is scanned and so on. This method requires an additional node length array to indicate the index of the last arc examined in each active star.

The second version of the LIFO procedure maintains the depth first character of the node scan but does not require the additional node length array of arc numbers. This version scans each labeled node completely--i.e., examines each element of its active star--before scanning the next node. We implemented this version in two ways, which we dubbed the modified LIFO scan and the threaded LIFO scan. The former simply selects the most recently labeled node (not yet scanned) as the node to scan next. The latter maintains the list of labeled nodes in thread order, and after scanning a labeled node, scans its thread successor. The thread must be revised each time the scan of a node succeeds in labeling at least one new node, since this must alter the scanned node's thread successor. This produces the situation in which the thread successor (hence the next node to be scanned) is the first node which was labeled by its predecessor node.

### 5.3.3 Label Re-Use

Ford and Fulkerson [15, 16, 17, 18] proposed erasing all labels after each flow augmentation step and initiating the next application of the labeling procedure anew at the source node. Later investigators [14] have shown that a portion of the labels created in the prior labeling phase may be retained, requiring only a subset to be erased.

The subset of labels to be erased may be identified in the following manner. For arcs in the flow augmentation path with an updated net capacity value of zero, the *first blocking arc* is defined to be that arc (u,v) which is closest to the source node. Assuming without loss of generality that arc (u,v) is directed away from the source, the subtree hanging from node v must be deleted from the label tree. This subtree is referred to as the *leaving subtree* while the remaining subtree which contains the source is called the *main subtree*.

It may be necessary, however, to rescan (or partly scan) some of the main subtree nodes. For example, it suffices to rescan main subtree nodes labeled after node v, thereby detecting if a flow eligible arc exists from one the these nodes to the leaving subtree. In the case of a FIFO node scan, the nodes contained in the leaving subtree cannot be determined efficiently. Consequently, when using the FIFO procedure, the labels for *all* nodes labeled after node v are erased and the node scan recommences at node u. (This procedure is simplified if node v is the terminal node. In that case, the label for node v is erased and the node scan recommences at node u.)

In the thread node scan, a thread trace is used to erase the labels of the nodes in the leaving subtree before recommencing the node scan at node u. An additional node length array (DEPTH) is used to record the depth of each labeled node so as to determine which nodes are in the leaving subtree.

More complex options for erasing and resetting labels exist, but they require the maintainence and updating of additional arrays that render these options unattractive.

### 5.3.4 Largest Augmentation

A third method used to scan the labeled nodes is referred to in the literature as the method of the largest (or maximum) augmentation [6, 12]. This method constructs a label tree by identifying and labeling a label-eligible node j for which the allowable flow change is maximal. A two-part label which records allowable flow changes as well as predecessors is essential in this case.

The largest augmentation method has been reported in the literature as computationally inefficient. However, our findings suggest that this conclusion is more a result of the previous implementations rather than of the method itself. For example, Cheung's implementation requires a node length pass of the second (flow change) part of the label array in order to select the next node to be scanned. This extensive processing of the label yields long solution times.

We propose rather that the label tree be constructed using an address calculation sort in which the addresses of the sort array correspond to allowable flow change values. This approach requires an additional array, SARRAY, which contains MAXS entries, where MAXS denotes the maximum arc capacity of the arcs in the active star of the source node. For each flow augmentation value d, $1 \le d \le$ MAXS, SARRAY(d) points to the next node having a flow augmentation value of d. These nodes appear in OVER, a node length overflow array. These two arrays, SARRAY and OVER, may be conceived as (potentially) forming MAXS lists which are processed in LIFO order. For example, if k is a node which has an allowable flow augmentation value of d and node j is now found to have the same value, this is indicated by setting SARRAY(d) = j and OVER(j) = k. An additional node length array, LABLF, can be utilized to maintain the second portion of each label, the current flow augmentation value for each node.

SARRAY is processed and the node scan is performed in the following way. At the beginning of each labeling step, all of the entries in SARRAY, LABLF, and OVER are set to zero. STEP 1 is completed by labeling the source node and setting its imputed flow change value to MAXS. (No labels are retained from the prior labeling step.) STEP 2 is performed by finding the largest index d for which SARRAY(d) (=i) is nonzero. Then node i is scanned. For each arc (i,j) in the complete star of i, the imputed allowable change value $d_1$, via each flow eligible arc (i,j) is computed. If the current value of LABLF(j) is $d_2$ and $d_1$ is greater than $d_2$, then LABLF(j) is reset to $d_1$ and node j is removed from SARRAY($d_2$) and added to SARRAY($d_1$). By construction, LABLF(i) will not be changed and is permanent after node i is scanned.

Then each of the nodes having the same allowable flow augmentation value as node i are scanned. When no more nodes have that value, a node having the next highest value is added to the label tree and the arcs in its complete star examined. The labeling procedure terminates when SARRAY is emptied. It is important to note that the sole purpose of LABLF is to facilitate the update of the temporary flow augmentation values.

While these refinements clearly improve upon the standard form of the largest flow augmentation procedure, we propose an additional refinement that results in what we call the *modified largest augmentation* procedure. In this approach, during the examination of each flow eligible arc in the complete star of the labeled node currently being scanned, each unlabeled endpoint receives a *permanent* label. The scan order of labeled nodes is unchanged: this modified method still chooses to scan the arcs in the complete star of a node with the largest flow augmentation value. However, only a single predecessor arc label is required and the labeling procedure terminates as soon as the terminal node is labeled.

The modified largest augmentation method will not, of course, always identify a path from source to terminal with maximum net capacity. However, it tends to examine fewer arcs per labeling. (It also requires one fewer node length array.) Our development and testing of this modified approach, in place of the standard largest flow augmentation approach, was motivated by the findings of other researchers that the standard largest augmentation procedure is inferior to other label tree procedures.

Three versions of the modified largest augmentation scan were developed. The first, referred to as MAXAUG, uses the SARRAY and OVER lists to scan the nodes in a straight-forward manner. It should be noted that the requirement that SARRAY is an MAXS length list effectively restricts the use of the MAXAUG scan to problems for which MAXS is small.

The fact that MAXS is infinity for the multi-terminal random and transit and network problems motivated the development of two refinements of the modified largest augmentation scanning procedure. The first of these, referred to as BUCAUG, can be thought of as using a MAXS + 1 length SARRAY list. The first MAXS entries in the list point to nodes which allow from 1 to MAXS units of flow change, and the last entry in the list points to nodes that allow over MAXS units of flow change. That is, the last entry corresponds to an infinite width bucket. This implementation reintroduces the list to maintain the allowable flow change values for those nodes which allow LABLF more than MAXS units of flow change.

The second refinement of the modified largest augmentation scanning procedure is referred to as MODAUG. Like the BUCAUG approach, MODAUG uses an infinite width bucket at the end of the SARRAY. However, to reduce storage requirements the LABLF array is not used. So, the exact amount of allowable flow change is unknown for any node in the last position of the SARRAY list. MODAUG, therefore, augments flow by MAXS + 1 units even if the arcs on a flow augmentation path could accommodate a larger increase.

## 5.4 Computational Testing

The initial phase of testing label tree implementations was devoted to determining the most efficient implementations for each of the node scans (FIFO, LIFO, and modified largest augmentation) and for each of the special data structures (FSLRS, FIXMIR, and DYNMIR). Then the performances of the resulting codes (and that of the sequential code using the MFS data structures) were evaluated with respect to the test problems.

### 5.4.1 Initial Testing Phase

Three computational refinements designed to improve code performance were analyzed. First, we examined the effects of label re-use schemes for FIFO and LIFO node scans. Our conclusions in this area coincide with those of earlier researchers [14]. The testing uniformly indicated the advisability of label re-use schemes although the actual savings in optimization time varies according to implementation and network problem topology. In some cases, the label re-use strategy is overwhelmingly superior. For example, re-using labels generated by the thread node scan on transit grid problems reduced optimization times by 80%.

We next investigated the length of SARRAY in the modified largest augmentation node scan for networks with large arc capacities (as in the multi-terminal random and transit grid problems). We implemented the three approaches discussed in Section 5.3 in the following fashion. The MAXAUG and BUCAUG node scans were implemented using the FSLRS structure and the MODAUG node scan was implemented using both the FIXMIR and DYNMIR structures. MAXS was set to be the largest ordinary (i.e., non-infinite) arc capacity.

The third general area of investigation involved the efficiency of schemes to partition the arc data. All three of the FSLRS, FIXMIR, and DYNMIR approaches performed better than the associated nonpartition methods. Using the FSLRS structures, nearly all of the arcs in the flow augmentation paths are labeled while examining the forward stars. Consequently, it became apparent that a two-phase or suboptimization approach should be developed. During the first phase the maximum flow problem is suboptimized by examining only the arcs in a node's forward star. Since the arcs of the forward star are stored in contiguous locations of the data arrays, processing a forward star is "easier" than processing a reverse star. The second phase of the optimization procedure considers the arcs in both a node's forward and reverse (i.e., complete) stars. This procedure greatly reduced the number of arc examinations. The partitions employed in the FIXMIR and DYNMIR structures similarly reduced the number of arc examinations because the initial zero flows on all original arcs yield zero net capacities for their mirrors, which are therefore not scanned.

## 5.4.2 Secondary Testing Phase

Since three node scan alternatives and four data structure alternatives were tested in the various implementations, we use the syntax A/B to distinguish among these implementations, where A identifies the type of node scan and B refers to the type of data structure employed.

The first phase of testing culled out the clearly inferior combinations and reduced the number of label tree codes under consideration to eleven. These include: a sequential access code, SEQUEN/MFS; three FIFO codes, FIFO/FSLRS, FIFO/FIXMIR, and FIFO/DYNMIR; three LIFO codes, LIFO/FSLRS, THREAD/FIXMIR, and THREAD/DYNMIR; and four modified largest augmentation codes, MAXAUG/FSLRS, BUCAUG/FSLRS, MODAUG/FIXMIR, and MODAUG/DYNMIR.

The optimization times and, where applicable, total solution times of these codes for the problem sets are presented in Table VI. (As discussed in Section 3.1, total solution time records the elapsed time after input of the network and prior to the solution output. Optimization time measures internal solution time and disregards the time required both to arrange the problem data in the function arrays and to retrieve the solution in a suitable output form.) For most of the problems, the node scan of a specific implementation affects optimization much more than does the set of data structures used to store the original arc data. Equivalently, the performance times and statistics of one implementation is indicative of those of other implementations which employ the same node scan and store the original arc data differently. So, we inserted counters into some of the codes (one per node scan method) and re-solved a subset of the problems in order to interpret code performance. For the sake of brevity and expositional clarity, the complete statistics are not presented here but may be obtained from the authors. Less detailed statistics are employed to help explain the results shown in Table VI. The next subsection discusses the efficiency of label tree implementations for the random and multi-terminal

TABLE VI

COMPUTER TIMES IN SECONDS FOR LABEL TREE METHODS ON CDC 6600*

| PROBLEM | SEQUEN/MFS | FIFO/FSLRS | FIFO/FIXMIR | FIFO/DYNMIR | LIFO/FSLRS | THREAD/FIXMIR | THREAD/DYNMIR | MAXAUG/FSLRS | BUCAUG/FSLRS | MODAUG/FIXMIR | MODAUG/DYNMIR |
|---|---|---|---|---|---|---|---|---|---|---|---|
| R1 | .09 (.12) | .08 (.13) | .06 (.13) | .06 | .24 (.29) | .27 (.34) | .25 | .03 (.08) | .04 (.09) | .03 (.10) | .03 |
| R2 | .15 (.20) | .16 (.22) | .14 (.24) | .11 | .58 (.64) | .38 (.48) | .36 | .08 (.14) | .11 (.17) | .08 (.18) | .08 |
| R3 | .31 (.37) | .30 (.38) | .28 (.39) | .22 | .83 (.91) | .95 (1.06) | .84 | .19 (.27) | .22 (.30) | .14 (.25) | .13 |
| R4 | .28 (.35) | .28 (.37) | .21 (.32) | .18 | 1.42 (1.51) | 1.17 (1.28) | 1.09 | .29 (.38) | DNR | .21 (.32) | .18 |
| R5 | .40 (.51) | .39 (.52) | .41 (.55) | .33 | 1.69 (1.82) | 2.02 (2.16) | 1.85 | .21 (.34) | DNR | .20 (.34) | .19 |
| R6 | .62 (.76) | .80 (.97) | .40 (.60) | .65 | 2.75 (2.92) | 3.77 (3.97) | 3.39 | .39 (.56) | DNR | .31 (.51) | .29 |
| R7 | DNR | DNR | DNR | .19 | DNR | DNR | DNR | DNR | DNR | .30 (.46) | .27 |
| R8 | DNR | DNR | DNR | .51 | DNR | DNR | DNR | DNR | DNR | .35 (.61) | .31 |
| R9 | DNR | DNR | DNR | .85 | DNR | DNR | DNR | DNR | DNR | .44 (.77) | .38 |
| R10 | DNR | DNR | DNR | .35 | DNR | DNR | DNR | DNR | DNR | .20 (.43) | .18 |
| R11 | DNR | DNR | DNR | .78 | DNR | DNR | DNR | DNR | DNR | .52 (.84) | .49 |
| R12 | DNR | DNR | DNR | 1.44 | DNR | DNR | DNR | DNR | DNR | .61 (1.01) | .53 |
| MR1 | .43 (.46) | .51 (.56) | .52 (.55) | .45 | 1.58 (1.63) | 3.38 (3.41) | 3.19 | CNR | .38 (.43) | .36 (.39) | .34 |
| MR2 | 1.08 (1.13) | 2.45 (2.51) | 2.70 (2.78) | 2.22 | 9.84 (9.90) | 9.36 (9.44) | 8.64 | CNR | 1.18 (1.24) | 1.06 (1.14) | 1.02 |
| MR3 | 2.50 (2.56) | 2.30 (2.38) | 2.41 (2.51) | 1.94 | 10.35 (10.43) | 11.91 (12.01) | 10.43 | CNR | 1.86 (1.94) | 1.64 (1.74) | 1.42 |
| MR4 | .72 (.77) | .94 (1.03) | .87 (.97) | .63 | DNR | DNR | DNR | CNR | .74 (.83) | .62 (.72) | .56 |
| MR5 | 1.93 (2.04) | 2.68 (2.81) | 2.65 (2.80) | 2.23 | DNR | DNR | DNR | CNR | 1.08 (1.21) | 1.27 (1.42) | 1.18 |
| MR6 | 1.93 (2.04) | 8.08 (8.25) | 8.30 (8.49) | 7.15 | DNR | DNR | DNR | CNR | 6.47 (6.64) | 5.84 (6.03) | 4.95 |
| MR7 | DNR | DNR | DNR | 1.54 | DNR | DNR | DNR | CNR | DNR | .73 (.85) | .61 |
| MR8 | DNR | DNR | DNR | 4.10 | DNR | DNR | DNR | CNR | DNR | 2.04 (2.28) | 1.79 |
| MR9 | DNR | DNR | DNR | 12.00 | DNR | DNR | DNR | CNR | DNR | 6.22 (6.50) | 5.12 |
| MR10 | DNR | DNR | DNR | 1.18 | DNR | DNR | DNR | CNR | DNR | .50 (.70) | .45 |
| MR11 | DNR | DNR | DNR | 5.68 | DNR | DNR | DNR | CNR | DNR | 1.71 (1.99) | 1.56 |
| MR12 | DNR | DNR | DNR | 17.05 | DNR | DNR | DNR | CNR | DNR | 10.55 (10.92) | 9.27 |
| TG1 | .74 (.79) | .94 (.99) | 1.02 (1.07) | .87 | 1.72 (1.77) | .30 (.35) | .27 | 1.90 (1.95) | 1.82 (1.87) | DNR | 1.41 |
| TG2 | .26 (.30) | 1.13 (1.19) | 1.24 (1.29) | 1.16 | 1.81 (1.87) | .51 (.56) | .44 | 2.42 (2.48) | 2.51 (2.57) | DNR | 1.83 |
| TG3 | 1.32 (1.36) | 3.15 (3.23) | 3.27 (3.37) | 2.78 | 4.89 (4.97) | .82 (.92) | .74 | 4.63 (4.71) | 4.84 (4.92) | DNR | 3.73 |
| TG4 | .69 (.77) | 2.96 (3.04) | 3.03 (3.13) | 2.58 | 3.12 (3.20) | .49 (.59) | .46 | 4.15 (4.23) | 4.14 (4.22) | DNR | 3.11 |
| TG5 | 2.95 (3.08) | 7.21 (7.36) | DNR | 6.43 | 11.20 (11.35) | 1.16 (1.30) | 1.09 | DNR | DNR | DNR | 8.33 |
| TG6 | 1.03 (1.18) | 7.77 (7.93) | DNR | 7.26 | 12.25 (12.41) | 2.27 (2.45) | 1.96 | DNR | DNR | DNR | 13.79 |
| TG7 | 4.10 (4.28) | 7.97 (8.18) | DNR | 6.93 | 9.94 (10.15) | .99 (1.18) | .93 | DNR | DNR | DNR | 6.94 |
| TG8 | 3.31 (3.47) | 16.13 (16.32) | DNR | 15.20 | 20.18 (20.37) | 2.14 (2.37) | 1.86 | DNR | DNR | DNR | 18.93 |
| H1 | .21 (.21) | .15 (.15) | .20 | .17 | .15 (.15) | .18 (.19) | .14 | .23 (.23) | .17 (.17) | DNR | .16 |
| H2 | 3.29 (3.32) | 2.41 (2.44) | 3.73 | 2.42 | 1.91 (1.94) | 2.12 (2.14) | 1.69 | 3.20 (3.23) | 2.18 (2.21) | DNR | 2.50 |
| H3 | DNR | 11.95 (12.01) | DNR | 11.80 | 8.82 (8.88) | 9.77 (9.84) | 7.66 | CNR | DNR | DNR | 12.02 |
| H4 | DNR | 37.36 (37.47) | DNR | 36.89 | 26.46 (26.57) | 29.74 (29.87) | 23.14 | CNR | DNR | DNR | 36.95 |
| H5 | DNR | 90.44 (90.61) | DNR | 88.88 | 62.61 (62.78) | 70.88 (71.36) | 54.77 | CNR | DNR | DNR | 88.37 |

* Times not shown in parentheses indicate optimization time, and times in parentheses indicate total solution times.

random test problems, while subsequent subsections cover the transit grid and hard problems.

## Random and Multi-Terminal Random Networks

Codes employing the FIXMIR data structures tend to exhibit the best *total solution times* for the random and multi-terminal random problems while the codes implementing the DYNMIR structures obtain the best *optimization times* (exclusive of translating randomly ordered arc data into the data formats required for solutions). (See Table VI.) From best to worst (for both total solution times and optimization times) the node scan methods rank as follows: modified largest augmentation, sequential, FIFO, and finally, LIFO.

The excellent performance of the modified largest augmentation method on the random problems stems from the extremely low number of labelings required by this approach. This, in turn, results from a high average flow increase per augmentation, which is twice the average increase for FIFO trees. The label trees generated by the modified largest augmentation method are smaller than those generated by the otehr label tree methods and, for the first two problems, the percentage of network arcs examined per labeling step exceeds only that of the thread trees.

SEQUEN/MFS is about 25% slower than the modified largest augmentation codes because the MFS data structure does not allow the algorithm to concentrate on arcs with labeled from nodes. Instead, many irrelevant arcs (e.g., connecting pairs of unlabeled nodes) must be accessed by SEQUEN/MFS in order to locate the few good arcs that enable additional nodes to be labeled.

The average label trees generated by the next to worst approach, the FIFO node scan, contain over 80% of the nodes. Although about half of the labels are retained, the average number of arcs examined per labeling exceeds that of the modified largest augmentation approach by 10% to 90%. The label trees are broader and much shallower than those of the modified largest augmentation method. However, the average flow increase per flow augmentation path is half that of the modified largest augmentation method, while the number of labelings increases from 25% to 100% and the total number of arcs examined more than doubles.

As the solution times indicate, trees generated by the worst approach, the LIFO scan, are constructed "too quickly." While the average number of arcs examined per labeling step is the best among the tested codes, the average flow augmentation path is very long and admits a very small average flow increase. Thus, the number of labelings is prohibitive.

The average label trees generated by the node scan methods for the multi-terminal random problems resemble those for the random problems. However, more nodes are labeled and more arcs are examined per labeling. While the average flow augmentations remain about the same, the total maximum flow values greatly exceed the respective values of the random problems. So all of the codes experience an order of magnitude increase in the number of labelings, number of arcs examined, and total solution times.

## Transit Grid Networks

Remarkably, the LIFO label tree implementation, which was the worst for the random and multi-terminal random problems, performs better on the transit grid network problems than the FIFO, modified largest augmentation, and sequential approaches.

Indeed, the LIFO strategy is well suited to the transit grid network topology. Its average label trees include less than 60% of the network nodes, and of these, over 75% of the labels are retained for the next labeling step. Together these two factors indicate that few arcs need to be examined during each labeling step--the percentage is less than 5% for each of the two small problems. These very low values arise because the thread node scan order is particularly suited to this topology. Not only are there many arcs entering the terminal node but the transit grid structure yields nearly identical label trees from one iteration to the next. Further, the average flow augmentation paths for each of the label tree methods are shorter than those of the two previous topologies and permit a substantial average flow increase.

Again, the sequential approach is the second best of the label tree implementations. This may be due to the numbering system used in constructing the transit grid networks. Specifically, the nodes can be pictured as being numbered from left to right in the grid. Since arcs connect adjacent grid nodes, the sequential node scan is able to systematically move from right to left through the network. Unlike the case with the random and multi-terminal random networks, SEQUEN/MFS does not encounter, in the transit grid problems, as many "useless" arcs connecting pairs of unlabeled nodes.

The FIFO approach appears less well suited to the transit grid network topology. The large number of highly capacitated arcs which connect the source node to the grid nodes produces shallow broad trees. On the one hand, the average FIFO flow augmentation paths for the transit grid problems are shorter and accommodate a much larger flow increase than those generated for the two previous problem topologies. Further, the extreme breadth of the trees tends to limit the number of erased labels: two-thirds of the labels are retained as opposed to one-half for the two previous topologies. As impressive as these figures may seem, those of the thread trees are much better. Further, the FIFO label trees contain many more nodes than those of the thread trees and, by the nature of the node scan, examine appreciably more arcs.

The modified largest augmentation implementation performs poorly on these problems. This approach examines many more arcs and labels more nodes than any other implementation. The effect is to yield the longest solution times of any of the label tree methods on the transit grid problems.

## Hard Networks

Ranked from best to worst on the hard problems, the label tree methods performed as follows: LIFO, FIFO, modified largest augmentation, and then sequential.

Since most flow augmentation paths contain only arcs having a unit net capacity, label re-use is not quite as effective as with the prior topologies. Two label tree methods, however, are rather effective in exploiting the initial order of the arc data. For each node i $\neq$ t, the first arc of the flow eligible arcs in the complete star is the arc (i,i+1). The remaining arcs in the active star immediately follow in ascending to-node order. The LIFO methods tend first to scan the opposite endpoints of the initial and the final arcs in the appropriate active stars. The arc data of these latter arcs are exchanged with newly saturated arcs in the partition update phase. The modified largest augmentation methods process the network arcs similarly. Of the arcs in the active star of node i $\neq$ t, only arc (i,i+1) may have a net capacity exceeding unity. Since the remaining arcs in the active star all have a unit net capacity value, their opposite endpoints share the same location in SARRAY. So, the latter arcs in the active star of node i($\neq$ t) are accessed after the arc (i,i+1) since common entries in SARRAY are stored in LIFO order. The shared characteristic of both arc scans is that arcs in the middle of the active stars are accessed last.

The FIFO node scan provides a comparison with these approaches since it does not advantageously exploit the arc data order. Of the three methods, FIFO examines the largest number of arcs and posts the largest solution times. The modified largest augmentation method examines the least number of arcs. However, each network contains only a few distinct capacity values and so SARRAY contains many zero values. The overhead in processing this list results in solution times nearly as long as those of the FIFO approach. While the LIFO approach examines more arcs than the modified largest augmentation method, its solution times are the best of any obtained by the label tree approaches. The poor performance of the sequential approach may be attributed to the fact that the MFS structure requires the examination of all arcs in each forward star whereas the partitioning approaches concentrate on the arcs that allow flow increase. Since the optimal solution to a hard problem corresponds to all arcs at their upper bound, the MFS structure is inappropriate.

## 6.0 REFERENT METHODS

### 6.1 Algorithm Features

Label tree approaches normally find only one flow augmentation path for each application of the labeling procedure; but there may exist several such paths which consist of arcs connecting the labeled nodes. Following the lead of E. A. Dinic [11], several researchers have presented algorithms which modify the label tree approach so that one application determines *all* flow augmentation paths containing the least number of arcs [6, 24, 28]. Rather than creating a label tree, the labeling procedure first creates a *label subnetwork*. Constructively, this subnetwork consists of all nodes labeled in the formation of a FIFO label tree and all arcs that are flow eligible from a node labeled at a depth d from the source to a node labeled at depth d + 1 from the source.

The label subnetwork may be viewed as arising by "filling in" a FIFO label tree with all flow eligible arcs that connect some node in the tree to another at a depth one greater. As a result of using a FIFO label sequence, the path to every node is a shortest path (by the depth measure), and every flow eligible arc from a "shallower" to a "deeper" node in the FIFO tree must automatically be in the label subnetwork (i.e., the deeper node must be exactly one deeper).

The *referent*, finally, is the portion of this subnetwork consisting only of the arcs which actually lie on paths from the source to the terminal. Once the depth labels are attached, the referent arcs may be identified by starting at the terminal and in a backward pass isolating the *referent star* (the subset of the active star contained in the referent) for each node encountered during a FIFO label sequence back to the source. A series of forward traces then send as much flow as possible through the referent.

This description of the referent approach corresponds essentially to Dinic's original proposal, as implemented in [6]. More recent variants introduce changes in flow augmentation sequences and other embellishments. A loose framework for describing the referent methods is as follows:

STEP 1: [LABEL SUBNETWORK CREATION]

a. [INITIALIZATION] Label s and create a label subnetwork $Lsn(N_L, A_L)$ such that $N_L = \{s\}$ and $A_L = \emptyset$. Set $L = \{s\}$.

b. [NODE SELECTION] If $L = \emptyset$, go to STEP 5. Select some node i from the set L of labeled and unscanned nodes. Reset $L = L - \{i\}$.

c. [SCAN NODE] Scan node i assigning labels to all label eligible nodes and add these nodes (and implicitly the appropriate flow eligible arcs) to the label subnetwork Lsn($N_L$,$A_L$) and add these nodes to L. If node t is unlabeled, go to STEP 1b.

STEP 2: [REFERENT CREATION] Identify those label subnetwork nodes which are on some flow augmentation path from s to t. These nodes are the referent nodes.

STEP 3: [FLOW AUGMENTATION] Process the referent arcs in the active star of each referent node until no more flow may be transmitted from s to t.

STEP 4: [ERASE REFERENT] Erase the referent. Go to STEP 1a.

STEP 5: [TERMINATE] Stop. A set of maximum flow values has been determined.

The primary strategic possibilities for creating efficient implementations of referent methods include the following considerations:
(a) the method employed in STEP 2 to identify the referent nodes
(b) the use of a partition scheme to expedite the performance of STEPS 1-3
(c) the procedure employed in STEP 3 to augment flow through the referent.
These considerations do not affect the particular referent chosen but materially affect the speed of its creation and processing.

## 6.2 Problem Data Storage

With the exception of the label contents, referent methods utilize the same data structures as label tree methods. However, as the following two subsections indicate, referent creation and processing ideally require that the referent arcs be accessed as readily from one arc endpoint as from the other. Although not fully exploited by previous implementations, the mirror arc concept is particularly well suited to these operations. Hence, we implemented referent methods using the FIXMIR and the DYNMIR data structures.

## 6.3 Label Subnetwork and Referent Creation

The referent approaches of [6, 11, 13, 24, 28] create the label subnetwork and referent by a two-part approach employing two labeling steps. A *forward labeling step* begins labeling at the source node and forms the label subnetwork. A *backward step*, starting at the terminal node, determines which of these labeled nodes and associated arcs are in the referent. (A third step processes the referent.) For the example network, if the arc data is stored using the FIXMIR or DYNMIR data structures, then the label subnetwork of Figure 10 results. The corresponding referent appears in Figure 11.

FIGURE 10

LABEL SUBNETWORK

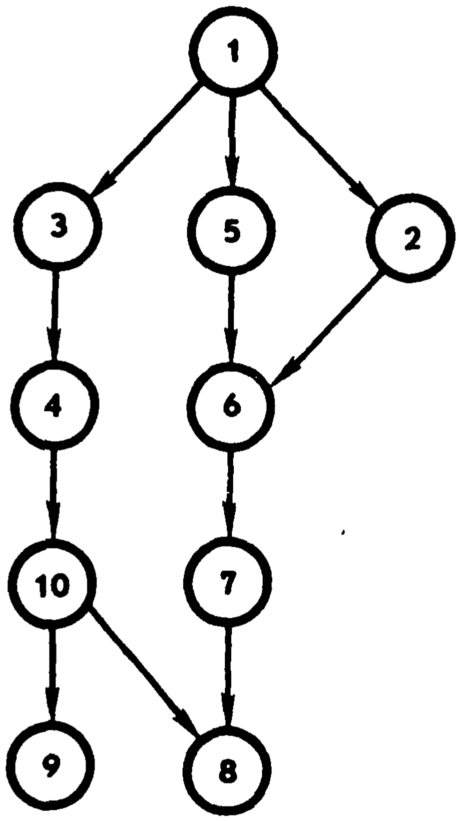
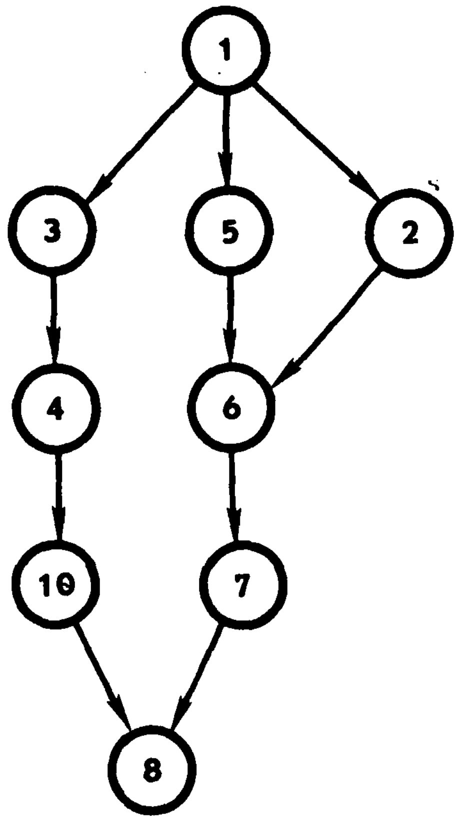


FIGURE 11

REFERENT



The use of the FIXMIR and DYNMIR data structures substantially accelerate the forward labeling step (which in some referent methods generates more--or different--information about the label subnetwork than in others). In the simplest form of the forward step, a label subnetwork is created by slightly modifying the procedure used to create FIFO label trees. The only difference is that depth labels rather than predecessor arc labels are used, where each labeled node is assigned a depth one greater (from the source) than the node scanned when the label was assigned. (Node s gets a depth of zero.) All otehr depth labels (for "unlabeled" nodes) are infinite.

Among the referent approaches appearing in the literature, Dinic's method [11] and Malhutra, Kumar, and Maheshwari's method [28] (hereafter referred to as the MKM method) have storage requirements comparable to label tree methods. Many of the other referent approaches are unattractive for large-scale applications since their implementation requires considerably more storage than do the label tree, Dinic's, or MKM's methods. Furthermore, Cheung compared the performance of his implementation of Dinic's method with some of these other methods [6], and found Dinic's approach to be uniformly superior from an empirical standpoint, in spite of the fact that some of the other methods have better worst case bounds. (The situation is reminiscent of the empirical performance of the primal simplex method against methods with superior theoretical bounds.) Consequently, only Dinic's method and the MKM method were used to provide the foundation of the approaches we tested.

## Dinic's Method

Dinic's original method is a simple three-pass algorithm in which the first two passes respectively create the depth labels and the referent stars, and the third pass seeks flow augmentation paths through the referent.

We did not implement Dinic's algorithm in this form, in spite of Cheung's findings that this produced the best of the methods he tested. The reason is that our use of the FIXMIR and DYNMIR data structures, which neither Dinic nor Cheung envisioned for application to maximum flow problems, not only streamlines the operations standardly performed, but makes alternative ways of processing the referent capable of being executed in highly economical ways. In particular, we propose new two types of methods derivative from the Dinic methodology: the *super-referent approach* and the *sub-referent* (or *implicit referent*) *approach.* Both of these referent approaches collapse the three-pass labeling process of Dinic's method into two passes, although each generates different types of information and carries out different operations on the way.

## The Super-Referent Method

The super-referent method utilizes the first pass to set up both distance labels and a super-referent star for each node encountered. The collection of all super-referent stars includes precisely the arcs of the label subnetwork, as earlier defined, but organized in *reverse fashion*, so they may be processed effectively on the second backward pass, without requiring that the network finally be traced again in a forward pass. The method is called the super-referent method because the structure it records is a superset of the referent structure recorded by Dinic's method.

To generate the super-referent stars, each arc that qualifies for inclusion in the label subnetwork--as an element of an active star whose opposite endpoint is deeper than the node currently scanned--is added to the super-referent list for the arc's opposite endpoint. This is done in the FIXMIR and DYNMIR data structures by accessing mirrors and re-arranging elements in the partition so that the super-referent star is in consecutive positions within the complete star. (An extra node length array is required for the additional "internal" partition. Alternatively, incurring a substantial penalty in increased memory, the process could be made slightly more efficient by allowing two additional arc length arrays to create the super-referent stars without re-arranging elements.)

When the backward pass is initiated from the terminal, only the super-referent star of each labeled node is processed during the node scan. However, the backward pass will now automatically scan only the referent nodes and arcs, because for each node actually in the referent (as contrasted with those only in the label subnetwork), the super-referent star will consist precisely of referent arcs. The remainder of the processing of the backward pass uses virtually the same pointers and logic as the sub-referent approach, which we now describe.

## The Sub-Referent Method

The sub-referent method does not attempt to set up any part of the referent at all on the initial forward pass, but--as in Dinic's original method--simply assigns distance labels to the nodes. However, the ensuing backward pass undertakes to do all the processing required to generate an "essential portion" of the referent. We call it the sub-referent method because the essential portion is typically only a proper subset of the full referent structure. More particularly, the backward pass immediately sets about identifying flow augmentation paths, where each step is organized in such a fashion that only referent paths are traced. In the process, some referent paths are automatically excluded because they are no longer augmenting paths by the time they are ready to be considered. Thus, this approach only implicitly traces the referent structure.

To describe the approach, we define the *reverse referent star* of a node to be the subset of referent arcs whose opposite endpoints have *smaller* (rather than larger) distance labels than the node itself. (The reverse referent star will coincide with the super-referent star for nodes in the referent.) Since this arc set is not identified on the forward pass of the sub-referent method, its members are culled on the backward pass. This culling is facilitated using the FIXMIR and DYNMIR structures by noting that an arc qualifies as a member if its *mirror* is in the forward star of the opposite node and if the distance label satisfies the previously indicated stipulation.

As soon as an element of a node's reverse referent star is identified, a special pointer identifies the position of this arc, whereon the scan of the node is temporarily discontinued, and the scan is immediately resumed at the opposite node for this referent arc. In this way the scan follows a strict LIFO pattern, moving from the terminal to the source. When the source is encountered the maximum allowable flow change is made, and the scan resumes at the node heading the *last* blocking arc (closest to the terminal), advancing the special pointer by at least one for each path arc whose net capacity drops to zero.

To avoid unnecessary tracing of dead end paths, whenever a referent star for a apricular node has been exhausted (which may be checked immediately when a net capacity is driven to zero, or deferred until the node is reached on a subsequent scan), this node may be "removed" from all reverse referent star lists simply by setting its distance label back to infinity. More extensive reverse scans to probe for sequences of nodes which may be disconnected in domino fashion are possible, but were not tested because they require expensive computer overhead with inadequate promise of compensation.

Backtracking occurs when there are no further reverse referent star elements for the node currently being scanned. This is accomplished by resetting the scanned node's distance label to infinity and in this case returning to the path predecessor of this scanned node, resuming in typical fashion. A backtracking step initiated from the terminal node results in termination of the entire process.

Comparative analyses of the steps of the super-referent and sub-referent approaches led us to anticipate that not only should both of these method be superior to the original Dinic approach, but that the sub-referent method should be superior to the super-referent method. The solution times reported in Table VII confirm this expectation.

## MKM's Method

The MKM algorithm augments flow through the referent in the following manner. The algorithm first determines the *flow potential* for each node in the referent. This value is the smaller of two sums of allowable flow increases; the sum for all referent arcs entering the node, and the sum for all referent arcs leaving the node. A node with minimum flow potential is determined. This node is called the *reference node* and its flow potential is termed the *reference potential*. The method then attempts to transmit a flow from the source to the terminal through the reference node equal to the reference potential. When the flow potential at some node i becomes zero, then all remaining referent arcs in FS(i) and RS(i) are removed from the referent and the updated flow potentials computed. Processing of the current referent terminates when the flow potential of every node is zero.

Since two passes are required to assign depth labels and to determine those nodes which are in the referent, neither the sub-referent nor the super-referent approach may be used to implement the MKM method. Further, the MKM flow augmentation approach is completely different from that of standard label algorithms. Instead of forming a flow augmentation path from the source to the terminal, the MKM algorithm forms a flow augmentation *sub-*

*network* from the source through the reference node to the terminal. This subnetwork is not generated via a labeling procedure and the amount of flow (reference potential) to be transmitted from the source to the terminal via this subnetwork is known prior to the generation of the subnetwork.

The disadvantage posed by this flow augmentation approach is that during the processing of a single referent, individual referent arcs may be contained in several flow augmentation subnetworks and hence be accessed several times. Our computational results indicate that these features proved to be a major limitation.

Efficient implementation of this approach requires several additional data structures. The SUMIN and SUMOUT arrays contain the sum of the arc net capacities for the referent arcs entering and leaving each referent node. (To ease the computational effort SUMIN(s) and SUMOUT(t) are respectively assigned the same values as SUMOUT(s) and SUMIN(t).) For each referent node, the flow potential over the referent arcs is equal to the minimum of the associated SUMIN and SUMOUT values. A node having the minimum flow potential value, REFPOT, is selected as the reference code. The REFPOT flow value, the reference potential, is then sent along the path from the source through the reference node to the terminal node.

The storage of the subnetwork arcs on which flow is changed (in a flow augmentation step) is completely different from the storage of flow augmentation path arcs. The nodes of the referent are stored in a node length array, REFER. All of the nodes at the same depth are grouped together so that nodes at the same depth in the subnetwork may be easily located. The amount of flow which must be sent through each of these nodes is stored in another node length array, NODFLO.

Flow is augmented in a two-part process. In the forward flow step, flow is sent from the reference node to the terminal. For each node n in this part of the subnetwork, NODFLO(n) units of flow must be moved forward to those nodes which are in the active forward star of n and are in the referent at the next level. The appropriate entries in REFER are flagged (if currently unflagged). After all NODFLO(n) units have been moved, then the next flagged node in REFER is processed. The backward flow step is analogous to the step forward. In this case a demand for REFPOT units of flow is transferred through the referent from the reference node back to the source node.

At the end of a flow augmentation step, the flow potential value for the reference node (and possibly other referent nodes) is now zero. This node and all referent arcs which either enter or leave it are removed from the referent by destroying the node's label. Removing these arcs may then reduce the flow potential value for some other referent nodes to zero. A final node length array, DELETE, is used to store these nodes until they can be removed from the referent. After all nodes are removed from the referent, the next referent is created.

## 6.4 Computational Testing

The computational testing for the referent approaches was performed in two parts. First, a preliminary version of the sub-referent approach was implemented. The difference between this version, called PRE, and the final version is that PRE does not employ a partition to isolate the arcs in each active star. The performance of PRE was compared to that of an implementation, called MKM, based on the approach of Malhutra, Kumar, and Maheshwari. MKM, like PRE, does not employ the active star partition. The final phase of testing compared the performance of the final version of the sub-referent method to an implementation of the super-referent method.

Both MKM and PRE were implemented using the DYNMIR data structures. Since both methods set up the distance labels identically on the forward pass, any differences in optimization time between the two methods may be attributed to differences in processing on subsequent passes.

The optimization times of these two codes for the problem sets are presented in Table VII. (As discussed in Section 3.1, total solution time records the elapsed time after input of the network and prior to the solution output. Optimization time measures internal solution time and disregards the time required to arrange the problem data in the function arrays and to retrieve the solution in a suitable output form.) The preliminary sub-referent approach is clearly superior to the MKM method, yielding smaller times for each problem set. Further, the difference becomes more pronounced as the density of the individual referents increases.

TABLE VII

COMPARISON OF LABEL TREE CODES
(Times in seconds on CDC 6600)*

| | PATH AUGMENTATION | | | | | | SUB-NETWORK AUGMENTATION |
|---|---|---|---|---|---|---|---|
| PROBLEM | PRE/FIXMIR | PRE/DYNMIR | SUP/FIXMIR | SUP/DYNMIR | SUB/FIXMIR | SUB/DYNMIR | MKM/DYNMIR |
| R1 | .11 ( .17) | .06 | .16 ( .23) | .13 | .05 ( .11) | .05 | .10 |
| R2 | .18 ( .25) | .12 | .25 ( .34) | .22 | .08 ( .16) | .06 | .18 |
| R3 | .11 ( .20) | .08 | .19 ( .28) | .16 | .10 ( .19) | .07 | .14 |
| R4 | .23 ( .34) | .13 | .29 ( .40) | .28 | .11 ( .21) | .08 | .17 |
| R5 | .41 ( .54) | .24 | .51 ( .64) | .47 | .09 ( .24) | .09 | .31 |
| R6 | .22 ( .40) | .16 | .59 ( .77) | .48 | .18 ( .39) | .15 | .27 |
| R7 | .27 ( .43) | .19 | DNR | .40 | .18 ( .32) | .13 | .24 |
| R8 | .23 ( .45) | .14 | DNR | .47 | .18 ( .40) | .20 | .22 |
| R9 | .47 ( .76) | .31 | DNR | .75 | .32 ( .62) | .21 | .51 |
| R10 | .74 ( .94) | .51 | DNR | .88 | .18 ( .38) | .15 | .56 |
| R11 | .71 ( 1.01) | .49 | DNR | .89 | .31 ( .59) | .25 | .57 |
| R12 | 1.00 ( 1.36) | .63 | DNR | 1.12 | .32 ( .68) | .29 | .74 |
| MR1 | .30 ( .35) | .22 | .30 ( .34) | .29 | .14 ( .18) | .13 | .32 |
| MR2 | .41 ( .48) | .32 | .43 ( .51) | .38 | .29 ( .37) | .24 | .85 |
| MR3 | .39 ( .48) | .28 | .44 ( .53) | .37 | .27 ( .36) | .22 | .79 |
| MR4 | .48 ( .58) | .31 | DNR | .48 | .20 ( .29) | .17 | .47 |
| MR5 | .56 ( .72) | .37 | DNR | .62 | .31 ( .46) | .27 | .85 |
| MR6 | .76 ( .95) | .55 | DNR | .80 | .50 ( .69) | .41 | 1.66 |
| MR7 | .61 ( .74) | .44 | DNR | .79 | .36 ( .48) | .29 | .68 |
| MR8 | .67 ( .89) | .48 | DNR | .91 | .41 ( .65) | .37 | 1.16 |
| MR9 | 1.37 ( 1.64) | 1.01 | DNR | 1.43 | .71 (1.00) | .65 | 2.92 |
| MR10 | 1.03 ( 1.23) | .72 | DNR | 1.15 | .30 ( .51) | .27 | .82 |
| MR11 | 1.31 ( 1.61) | .90 | DNR | 1.41 | .50 ( .78) | .45 | 1.56 |
| MR12 | 1.35 ( 1.70) | .92 | DNR | 1.26 | .80 (1.25) | .60 | 3.57 |
| TG1 | .37 ( .41) | .27 | .41 ( .45) | .37 | .28 ( .32) | .21 | .70 |
| TG2 | .25 ( .30) | .19 | .32 ( .37) | .27 | .21 ( .27) | .17 | .60 |
| TG3 | .68 ( .78) | .52 | .74 ( .83) | .70 | .50 ( .60) | .41 | 1.68 |
| TG4 | .63 ( .72) | .47 | .75 ( .85) | .59 | .39 ( .49) | .31 | 1.79 |
| TG5 | 1.21 ( 1.34) | .90 | DNR | 1.26 | .78 ( .92) | .63 | 3.54 |
| TG6 | .65 ( .82) | .50 | DNR | .66 | .69 ( .85) | .49 | 2.75 |
| TG7 | 2.20 ( 2.39) | 1.61 | DNR | 2.06 | .96 (1.16) | .78 | 4.77 |
| TG8 | 1.76 ( 1.98) | 1.26 | DNR | 1.82 | 1.08 (1.30) | .83 | 5.97 |
| H1 | .17 ( .18) | .15 | .15 ( .15) | .12 | .17 ( .17) | .13 | .29 |
| H2 | 1.47 ( 1.48) | 1.33 | 1.19 (1.20) | .98 | 1.28 (1.29) | .96 | 2.59 |
| H3 | 5.05 ( 5.12) | 4.59 | 4.01 (4.08) | 3.28 | 4.19 (4.26) | 3.18 | 9.54 |
| H4 | 12.13 (12.26) | 10.98 | DNR | 7.64 | 9.92(10.06) | 7.51 | 24.52 |
| H5 | 23.76 (23.96) | 21.46 | DNR | 14.98 | 19.39(19.58) | 14.48 | 51.43 |

*Total solution time is shown within parentheses.

This result is independent of the underlying network problem topology and indicates the relative inefficiency of the MKM flow augmentation approach. Not only does the MKM approach require several more lists to augment additional flow from the source to the terminal but individual racs may be examined many times. By contrast, after the forward pass that assigns distance labels, the sub-referent approach "examines"--or more precisely, stops the pointer at--each arc in the complete star of a referent node at most once.

This ability of the sub-referent approach to restrict its examination of arcs appears to be critical. To check this hypothesis, we also implemented and partially tested a variant which also applies only distance labels on the forward pass but utilizes a FIFO node scan to process the reverse referent stars on the backward pass. The resulting optimization times were extremely poor.

Based on the findings of these initial test comparisons, we developed an improved version of the PRE implementation which utilize the active star partition. We then tested and compared this version, entitled SUB, with SUP, an implementation of the super-referent method.

The results in Table VII indicate that the swapping of arc indexes (in the case of the FIXMIR data structures) or arc data (in the case of the DYNMIR data structures) does not enhance algorithmic performance over most problem topologies. Indeed, this approach is useful only for very special network topologies such as the hard problems.

Isolating the active star produces the most efficient implementation in terms of both optimization time and total solution time. Significantly fewer arcs are examined during label subnetwork creation; this is indicated by the excellent code performance as the number of nodes and arcs increases. In fact, the increase in optimization times corresponding to an increase in either of these parameters is surprisingly slight.

## 7.0 CONCLUSIONS

This section presents a comparison of seven of the best implementations of the basic maximum flow solution methods. Specifically, from

Section 4.4, SEQCS was selected as the best all-around primal simplex maximum flow code. Four label tree codes were selected from Section 5.4. The first two, FIFO/FIXMIR and FIFO/DYNMIR, incorporate the "collective wisdom" of contributors to the original label tree school of thought. These two codes make use of the best refinements [12, 14, 27] of the original algorithm of Ford and Fulkerson [15, 16, 17, 18]. The next two codes, MODAUG/FIXMIR and MODAUG/DYNMIR, are the best overall label tree codes. Finally, two codes from Section 6.4, SUB/SIXMIR and SUB/DYNMIR, were selected as the best implementation of the referent maximum flow algorithm.

Comparisons between different codes can be made on three grounds: (1) computer memory requirements, (2) optimization time, and (3) total solution time. The best code, with respect to each criterion, is reasonably clear for a given problem topology. However, each criterion yields a different winner. Consequently, researchers and practitioners must base their choice of algorithmic approach on their personal evaluation of the relative importance of these criteria in a given setting.

Table VIII presents the amount of core storage required to store the problem data for each of the seven codes. Clearly, the primal code is the best code in terms of this criterion since it requires roughly one-half to one-third of the core of the other codes. The superiority of the primal approach is even more pronounced when the maximum flow problem is embedded as a subproblem in a more general solution system. In this case the relative value of computer memory is increased since the maximum flow subproblem must share memory with the master problem and possibly other subproblems.

Table IX presents the optimization times and total solution times (in parenthesis) for each of the seven codes. Total solution time is the optimization time plus the time required to sort the random problem data into the appropriate order. That is, for the primal code total solution time includes the time to sort the arcs into forward star form. For the codes using the FIXMIR structure, total solution time includes the time necessary to set up the fixed mirror data structure. For both the primal and the FIXMIR codes, total solution time includes the time required to re-sort the arc flows into the original problem order. Total solution time is not reported for codes using the DYNMIR structure since these codes can only

TABLE VIII

MEMORY REQUIREMENTS

| CODE | NUMBER OF ARRAYS REQUIRED | | |
|---|---|---|---|
| | \|N\| | \|A\| | MAXS |
| SEQCS | 7 | 2 | 0 |
| FIFO/FIXMIR | 4 | 6 | 0 |
| FIFO/DYNMIR | 4 | 6 | 0 |
| MODAUG/FIXMIR | 5 | 6 | 1 |
| MODAUG/DYNMIR | 5 | 6 | 1 |
| SUB/FIXMIR | 5 | 6 | 0 |
| SUB/DYNMIR | 5 | 6 | 0 |

TABLE IX

COMPARISON OF BEST CODES

(TIMES SHOWN IN SECONDS ON CDC 6600)*

| PROBLEM | SEQCS | FIFO/FIXMIR | FIFO/DYNMIR | MODAUG/FIXMIR | MODAUG/DYNMIR | SUB/FIXMIR | SUB/DYNMIR |
|---|---|---|---|---|---|---|---|
| R1 | .08 ( .11) | .06 ( .13) | .06 | .03 ( .10) | .03 | .05 ( .11) | .05 |
| R2 | .18 ( .23) | .14 ( .24) | .11 | .08 ( .18) | .08 | .08 ( .16) | .06 |
| R3 | .16 ( .22) | .28 ( .39) | .22 | .14 ( .25) | .13 | .10 ( .19) | .07 |
| R4 | .16 ( .23) | .21 ( .32) | .18 | .21 ( .32) | .18 | .11 ( .21) | .08 |
| R5 | .33 ( .44) | .41 ( .55) | .33 | .20 ( .34) | .19 | .09 ( .24) | .09 |
| R6 | .52 ( .66) | .40 ( .60) | .65 | .31 ( .51) | .29 | .18 ( .39) | .15 |
| R7 | .27 ( .38) | DNR | .19 | .30 ( .46) | .27 | .18 ( .32) | .13 |
| R8 | .48 ( .65) | DNR | .51 | .35 ( .61) | .31 | .18 ( .40) | .20 |
| R9 | .84 (1.07) | DNR | .85 | .44 ( .77) | .38 | .32 ( .62) | .21 |
| R10 | .46 ( .63) | DNR | .35 | .20 ( .43) | .18 | .18 ( .38) | .15 |
| R11 | .69 ( .91) | DNR | .78 | .52 ( .84) | .49 | .31 ( .59) | .25 |
| R12 | 1.45 (1.72) | DNR | 1.44 | .61 ( 1.01) | .53 | .32 ( .68) | .29 |
| MR1 | .26 ( .29) | .52 ( .55) | .45 | .36 ( .39) | .34 | .14 ( .18) | .13 |
| MR2 | .58 ( .63) | 2.70 (2.78) | 2.22 | 1.06 ( 1.14) | 1.02 | .29 ( .37) | .24 |
| MR3 | .56 ( .62) | 2.41 (2.51) | 1.94 | 1.64 ( 1.74) | 1.42 | .27 ( .36) | .22 |
| MR4 | .28 ( .35) | .87 ( .97) | .63 | .62 ( .72) | .56 | .20 ( .29) | .17 |
| MR5 | .83 ( .94) | 2.65 (2.80) | 2.23 | 1.27 ( 1.42) | 1.18 | .31 ( .46) | .27 |
| MR6 | 1.49 (1.63) | 8.30 (8.49) | 7.15 | 5.84 ( 6.03) | 4.95 | .50 ( .69) | .41 |
| MR7 | .64 ( .75) | DNR | 1.54 | .73 ( .85) | .61 | .36 ( .48) | .29 |
| MR8 | 1.00 (1.17) | DNR | 4.10 | 2.04 ( 2.28) | 1.79 | .41 ( .65) | .37 |
| MR9 | 2.52 (2.75) | DNR | 12.00 | 6.22 ( 6.50) | 5.12 | .71 (1.00) | .65 |
| MR10 | .70 ( .87) | DNR | 1.18 | .50 ( .70) | .45 | .30 ( .51) | .27 |
| MR11 | 1.97 (2.19) | DNR | 5.68 | 1.71 ( 1.99) | 1.56 | .50 ( .78) | .45 |
| MR12 | 5.39 (5.67) | DNR | 17.05 | 10.55 (10.92) | 9.27 | .80 (1.25) | .60 |
| TG1 | .21 ( .26) | 1.02 (1.07) | .87 | DNR | 1.41 | .28 ( .32) | .21 |
| TG2 | .16 ( .20) | 1.24 (1.29) | 1.16 | DNR | 1.88 | .21 ( .27) | .17 |
| TG3 | .48 ( .54) | 3.27 (3.37) | 2.78 | DNR | 3.73 | .50 ( .60) | .41 |
| TG4 | .41 ( .49) | 3.03 (3.13) | 2.58 | DNR | 3.13 | .39 ( .49) | .31 |
| TG5 | .73 ( .86) | DNR | 6.43 | DNR | 8.33 | .78 ( .92) | .63 |
| TG6 | .58 ( .73) | DNR | 7.26 | DNR | 13.79 | .69 ( .85) | .49 |
| TG7 | .96 (1.14) | DNR | 6.93 | DNR | 6.94 | .96 (1.16) | .78 |
| TG8 | 1.12 (1.28) | DNR | 15.20 | DNR | 18.93 | 1.08 (1.30) | .83 |
| H1 | .06 ( .06) | .20 | .17 | DNR | .18 | .17 ( .17) | .13 |
| H2 | .43 ( .46) | 3.73 | 2.42 | DNR | 2.50 | 1.28 (1.29) | .96 |
| H3 | 1.39 (1.45) | DNR | 11.80 | DNR | 12.02 | 4.19 (4.26) | 3.18 |
| H4 | 3.22 (3.31) | DNR | 36.89 | DNR | 36.95 | 9.92(10.06) | 7.51 |
| H5 | 6.16 (6.29) | DNR | 88.88 | DNR | 88.37 | 19.39(19.58) | 14.48 |

*Total solution times shown in parentheses.

output the solution for the arcs arranged in the sequence that the code finally generates, and cannot recover the original arc sequence. (To allow such recovery would require additional array space as well as additional processing.)

In terms of best optimization time the sub-referent approach SUB/DYNMIR is the clear winner. The next best approach is our primal simplex variant. It is interesting to note that members of the standard approach (i.e., label tree) are almost always dominated by the best primal and referent methods.

The value of a code as a subroutine is better reflected by total solution time, since this time includes the time to set up the data structure and return the solution (in the original data order). The DYMMIR structure is no longer so attractive in this context--as when the maximum flow problem is embedded as a subproblem in a larger solution system--due to the obstacle it presents to recovering the original problem order. Instead, the best code in terms of minimum total solution time is the SUB/SIXMIR implementation of the sub-referent method. It is important to note, however, that theprimal simplex total solution time is equal to SUB/FIXMIR on the transit grid problems. Thus, in view of the reduced memory requirements of the primal approach, for transit planning purposes SEQCS may be reasonably regarded as the best all around code. Indeed, for investigations involving determinations of maximum flows between numerous source-terminal pairs in the same network, the primal code may have an additional advantage over the other approaches. Specifically, the primal code only has to absorb the set-up cost once, whereas the other codes must incur this cost for each source-terminal pair.

The two most important limitations of the study are: (1) the largest problem solved only had ten thousand arcs; and (2) all test problems were artificially generated. The first limitation is due to memory space limitations of the computed on which our tests were conducted, which allows a user to access seventy thousand words of central memory. The requirement of six arc length arrays for all of the label tree and referent method implementations resulted in the ten thousand arcs limit. The primal codes, which only use two arc length arrays, could of course solve substantially larger problems on the same machine.

The second limitation (artificial test problems) was due to the present inaccessibility of practical data for maximum flow problems, though this inaccessibility promises to be removed in the near future. However, we sought to minimize this limitation by designing the data base of test problems to reflect some of the network topologies encountered in real-world problems.

We envision two important types of extensions of this study: (1) developing codes with re-start capabilities, and (2) developing codes to handle very large problems by means of in-core out-of-core processing.

The re-start capability is important when the maximum flow between multiple source-terminal pairs must be determined for the same network. This capability would also be useful in planning studies involving multiple sets of arc capacities for the same network. The in-core out-of-core capability is important for the very large scale planning problems faced by governmental transportation planners. It also is valuable to some degree for solving moderate size problems on small to medium size computers. Future investigations of codes embodying these capabilities would be highly useful.

## REFERENCES

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1975.

[2] R. D. Armstrong, D. Klingman, and D. Whitman, "Implementation and Analysis of a Variant of the Dual Method for the Capacitated T ans-shipment Problem," Research Report CCS 324, Center for Cybernetic Studies, The University of Texas at Austin, 1978. To appear in *EJOR*.

[3] R. Barr, J. Elam, F. Glover, and D. Klingman, "A Network Augmenting Path Basis Algorithm for Transshipment Problems," Research Report CCS 272, Center for Cybernetic Studies, The University of Texas at Austin. To appear in *An International Symposium Volume on Extremal Methods and Systems Analysis.*

[4] R. Barr, F. Glover, and D. Klingman, "Enhancements of Spanning Tree Labeling Procedures for Network Optimization," *INFOR* 17, 1 (1979) 16-34.

[4a] R. S. Barr, F. Glover, and D. Klingman, "An Improved Version of the Out-of-Kilter Method and a Comparative Study of Computer Codes," *Mathematical Programming* 7, (1974) 60-86.

[5] G. Bayer, "MAXFLOW, ACM Algorithm 324," *Communications of the ACM*, 11 (1968) 117.

[6] T. Cheung, "Computational Comparison of Eight Methods for the Maximum Network Flow Problem," Technical Report 78-07, Department of Computer Sciences, University of Ottawa, Ottawa, Ontario, 1978.

[7] W. H. Cunningham, "A Network Simplex Method." *Mathematical Programming*, 11 (1976) 105-116.

[8] W. H. Cunningham, "Theoretical Properties of the Network Simplex Method." *Mathematics of Operations Research*, 4 (1979) 196-208.

[9] G. B. Dantzig and D. R. Fulkerson, "On the Max-Flow Min-Cut Theorem of Networks." *Annals of Mathematical Studies*, Princeton University Press, Princeton, N.J. (1956) 215-221.

[10] R. Dial, F. Glover, D. Karney, and D. Klingman, "A Computational Analysis of Alternative Algorithms and Labeling Techniques for Finding Shortest Path Trees." *Networks*, (1979) 215-248.

[11] E. A. Dinic, "Algorithm for Solution of a Problem of Maximum Flow in a Network with Power Estimation." *Soviet Math. Doklady*, 11 (1970) 1277-1280.

[12] J. Edmonds and R. M. Karp, "Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems," *Journal of the Association for Computing Machinery*, 19 (1972) 248-264.

[13] S. Even and R. E. Tarjan, "Network Flow and Testing Graph Connectivity." *SIAM Journal of Computing*, 4 (1975) 507-518.

[14] C. O. Fong and M. R. Rao, "Accelerated Labeling Algorithms for the Maximal Flow Problem with Applications to Transportation and Assignment Problems." Working Paper 7222, Graduate School of Business, University of Rochester (1974).

[15] L. R. Ford and D. R. Fulkerson, "Maximal Flow Through a Network." *Canadian Journal of Mathematics*, 8 (1956) 399-404.

[16] L. R. Ford and D. R. Fulkerson, "A Simple Algorithm for Finding Maximal Network Flows and an Application to the Hitchcock Problem." *Canadian Journal of Mathematics*, 9 (1957) 210-218.

[17] L. R. Ford and D. R. Fulkerson, *Flows in Networks*. Princeton University Press, Princeton, N.J. (1962).

[18] D. R. Fulkerson and G. B. Dantzig, "Computations of Maximal Flows in Networks," *Naval Research Logistics Quarterly*, 2 (1955) 277-283.

[19] J. Gilsinn and C. Witzgall, "A Performance Comparison of Labeling Algorithms for Calculating Shortest Path Trees." NBS Tech. Note 772, U.S. Department of Commerce (1973).

[20] F. Glover, D. Karney, and D. Klingman, "A Computational Study on Start Procedures, Basis Change Criteria, and Solution Algorithms for Transportation Problems," *Management Science*, 20 (1974) 793-813.

[21] F. Glover, D. Karney, and D. Klingman, "Implementation and Computational Comparisons of Primal, Dual, and Primal-Dual Computer Codes for Minimum Cost Network Flow Problems," *Networks*, 4 (1974) 191-212.

[22] R. Helgason, J. Kennington, and J. Hall, "Primal Simplex Network Codes: State-of-the-Art Implementation Technology," Technical Report IEOR 76014, Department of Industrial Engineering and Operations Research, Southern Methodist University (1976).

[23] E. L. Johnson, "Networks and Basic Solutions," *Operations Research*, 14 (1966) 619-623.

[24] A. V. Karzanov, "Determining the Maximal Flow in a Network by the Method of Preflows." *Soviet Math. Doklady*, 15 (1972) 434-437.

[25] B. Kinariwala and A. G. Rao. "Flow Switching Approach to the Maximum Flow Problem: I." *Journal of the Association for Computing Machinery*, 24 (1977) 630-645.

[26] D. Klingman, J. Mote, and D. Whitman, "Improving Flow Management and Control Via Improving Shortest Path Analysis," Research Report CCS 322, Center for Cybernetic Studies, The University of Texas at Austin, 1978.

[27] P. M. Lin and B. J. Leon, "Improving the Efficiency of Labeling Algorithms for Maximum Flow in Networks," *Proceedings IEEE International Symposium on Circuits and Systems*, (1974) 162-166.

[28] V. M. Malhutra, M. P. Kumar, and S. N. Maheshwari, "An $O(|V|^3)$ Algorithm for Finding Maximum Flows in Networks." *Information Processing Letters*, 7, 6 (1978) 277-278.

[29] J. Mulvey, "Column Weighting Factors and Other Enhancements to the Augmented Threaded Index Method for Network Optimization," to appear in *Mathematical Programming*.

[30] A. Nijenhuis and H. S. Wilf. *Combinatorial Algorithms*, Academic Press (1975) 143-151.

[31] S. Phillips and M. I. Dessouky. "The Cut Search Algorithm with Arc Capacities and Lower Bounds." *Management Science*, 25 (1979) 396-404.

[32] V. Srinivasan and G. L. Thompson, "Accelerated Algorithms for Labeling and Relabeling Trees with Applications to Distribution Problems," *Journal of the Association for Computing Machinery*, 19 (1972) 712-726.

[33] V. Srinivasan and G. L. Thompson, "Benefit-Cost Analysis of Coding Techniques for the Primal Transportation Algorithms," *Journal of the Association for Computing Machinery*, 20 (1973) 194-213.

[34] N. Zaden, "Theoretical Efficiency of the Edmonds-Karp Algorithm for Computing Maximal Flows." *Journal of the Association for Computing Machinery*, 19 (1972) 184-192.